



Java SE: Programming I

Activity Guide

D102470GC20

Edition 2.0 | January 2019 | D105824

Learn more from Oracle University at education.oracle.com

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

For Instructor Use Only.
This document should not be distributed.

Table of Contents

| | |
|--|-----------|
| Practices for Lesson 1: Introduction..... | 5 |
| Practices for Lesson 1 | 6 |
| Practices for Lesson 2: What Is a Java Program?..... | 7 |
| Practices for Lesson 2 | 8 |
| Practices for Lesson 3: Creating a Java Main Class | 9 |
| Practices for Lesson 3 | 10 |
| Practices for Lesson 4: Data in a Cart..... | 11 |
| Practices for Lesson 4 | 12 |
| Practices for Lesson 5: Managing Multiple Items..... | 13 |
| Practices for Lesson 5 | 14 |
| Practices for Lesson 6: Describing Objects and Classes | 15 |
| Practices for Lesson 6: Overview | 16 |
| Practice 6-1: Creating Classes for the Soccer League..... | 17 |
| Practice 6-2: Creating a Soccer Game | 24 |
| Practices for Lesson 7: Manipulating and Formatting the Data in Your Program..... | 25 |
| Practices for Lesson 7: Overview | 26 |
| Practice 7-1: Manipulating Text..... | 27 |
| Practices for Lesson 8: Creating and Using Methods | 31 |
| Practices for Lesson 8: Overview | 32 |
| Practice 8-1: Using Methods | 33 |
| Practice 8-2: Creating Game Data Randomly..... | 35 |
| Practice 8-3: Creating Overloaded Methods | 40 |
| Practices for Lesson 9: Using Encapsulation..... | 43 |
| Practices for Lesson 9: Overview | 44 |
| Practice 9-1: Encapsulating Attributes..... | 45 |
| Practice 9-2: Adding Constructors..... | 47 |
| Practices for Lesson 10: More on Conditionals..... | 53 |
| Practices for Lesson 10: Overview | 54 |
| Practice 10-1: Using Conditionals | 55 |
| Practice 10-2: Debugging..... | 60 |
| Practices for Lesson 11: Working with Arrays, Loops, and Dates..... | 67 |
| Practices for Lesson 11: Overview | 68 |
| Practice 11-1: Iterating Through Data..... | 69 |
| Practice 11-2: Working with LocalDateTime | 73 |
| Practices for Lesson 12: Using Inheritance | 77 |
| Practices for Lesson 12: Overview | 78 |

| | |
|--|------------|
| Practice 12-1: Creating a Class Hierarchy | 79 |
| Practice 12-2: Add a GameEvent Hierarchy | 82 |
| Practices for Lesson 13: Using Interfaces | 85 |
| Practices for Lesson 13: Overview | 86 |
| Practice 13-1: Overriding the <code>toString</code> Method | 87 |
| Practice 13-2: Implementing an Interface | 89 |
| Practice 13-3: Using a Lambda Expression for Sorting (Optional Practice) | 92 |
| Practices for Lesson 14: Handling Exceptions | 95 |
| Practices for Lesson 14: Overview | 96 |
| Practice 14-1: Overview – Adding Exception Handling | 97 |
| Practices for Lesson 15: Deploying and Maintaining the Soccer Application | 101 |
| Practices for Lesson 15..... | 102 |
| Practices for Lesson 16: Understanding Modules..... | 103 |
| Practices for Lesson 16: Overview | 104 |
| Practice 16-1: Creating a Modular Application from the Command Line..... | 105 |
| Practice 16-2: Compiling Modules from the Command Line..... | 108 |
| Practice 16-3: Creating a Modular Application from NetBeans..... | 110 |
| Practices for Lesson 17: JShell..... | 113 |
| Practices for Lesson 17: Overview | 114 |
| Practice 17-1: Variables in JShell..... | 115 |
| Practice 17-2: Methods in JShell | 116 |
| Practice 17-3: Forward-Referencing..... | 118 |

Practices for Lesson 1: Introduction

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 1

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 2: What Is a Java Program?

Practices for Lesson 2

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 3: Creating a Java Main Class

Practices for Lesson 3

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 4: Data in a Cart

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 4

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 5: Managing Multiple Items

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 5

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 6: Describing Objects and Classes

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 6: Overview

Practices Overview

In these practices, you will create the classes needed to represent the data for the soccer application and write code to instantiate these classes.

For Instructor Use Only.
This document should not be distributed.

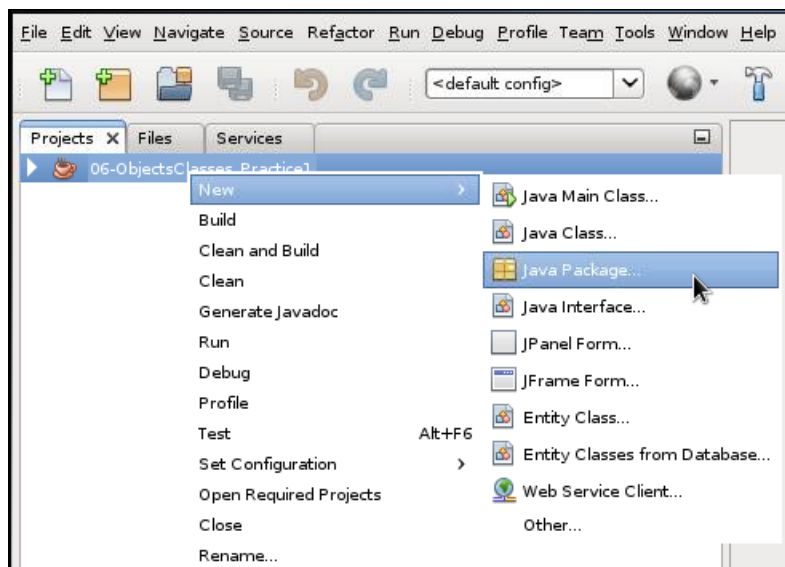
Practice 6-1: Creating Classes for the Soccer League

Overview

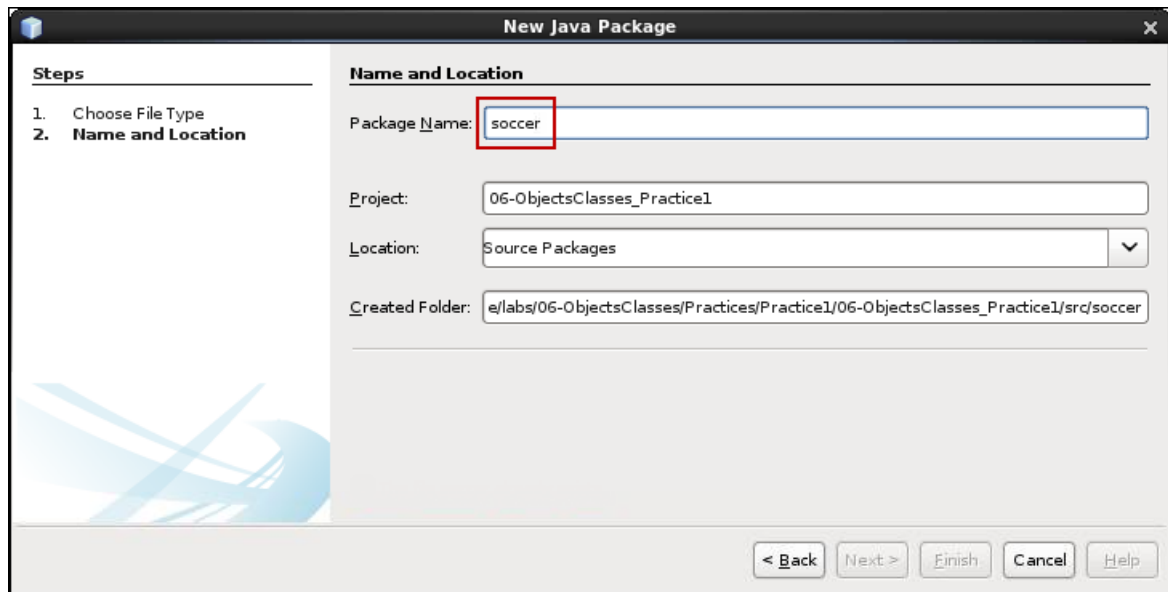
In this practice, you create the five classes required for the soccer application (Goal, Game, Player, Team, and League).

Tasks

1. Open the project for this practice in NetBeans.
 - a. Start NetBeans by double-clicking the NetBeans icon on the desktop.
 - b. Open the project **06-ObjectsClasses_Practice1** by clicking the Open Project icon, and then navigate to
~\labs\06-ObjectsClasses\Practices\Practice1\06-ObjectsClasses_Practice1
and click Open Project.
2. Create a package named `soccer` in the **06-ObjectsClasses_Practice1** project.
 - a. In NetBeans, right-click the project named **06-ObjectsClasses_Practice1**, and select New > Java Package.



- b. In the New Java Package dialog box, enter `soccer` as the Package Name, and click Finish.



3. The initial version of the soccer application comprises five main data classes: `Player`, `Team`, `Game`, `Goal`, and `League`. Create the `Player` class.
- Right-click the `soccer` package and select `New > Java Class`.
 - In the New Java Class dialog box, enter `Player` as the Class Name and click Finish.
 - In the most basic version of the soccer application, the `Player` class has only one attribute, a `String` called `playerName`. Add this attribute now.
 - The code of the `Player` class should now look like this:

```
package soccer;
public class Player {
    public String playerName;
}
```

4. Create the `Team` class.
- Right-click the `soccer` package and select `New > Java Class`.
 - In the New Java Class dialog box, enter `Team` as the Class Name and click Finish.
 - In the most basic version of the soccer application, the `Team` class has two attributes: a `String` named `teamName` and a `Player` array named `playerArray`. Add these attributes now.
 - The code of the `Team` class should now look like this:

```
package soccer;
public class Team {
    public String teamName;
    public Player[] playerArray;
}
```

5. Create the `Goal` class with the following attributes.

| Type | Name |
|--------|------------------------|
| Team | <code>theTeam</code> |
| Player | <code>thePlayer</code> |
| double | <code>theTime</code> |

The code of the `Goal` class should look like this.

```
package soccer;
public class Goal {
    public Team theTeam;
    public Player thePlayer;
    public double theTime;
}
```

6. Create the `Game` class with the following attributes:

| Type | Name |
|---------------------|-----------------------|
| Team | <code>homeTeam</code> |
| Team | <code>awayTeam</code> |
| <code>Goal[]</code> | <code>goals</code> |

The code of the `Game` class should look like this.

```
package soccer;
public class Game {
    public Team homeTeam;
    public Team awayTeam;
    public Goal[] goals;
}
```

7. Now that you have created the required classes, you create one more class called `League`, which will use these classes to run a set of games. This class needs to have a method `main` so that it can be run as a console application.
- Right-click the `soccer` package and select `New > Other`.
 - In the `New File` dialog box, select `Categories > Java`, then select `File Types: > Java Main Class`, and click `Next`.

- c. In the New Java Class dialog box, enter `League` as the Class Name and click Finish. Your new class should look like this (there will be other comments in your code that are not shown here).

```
package soccer;

public class League {
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

8. Using the classes that you have created means populating them with data. So you need to:

- Instantiate a number of `Player` objects for each `Team`.
- Instantiate some `Team` objects to play games.
- Instantiate some `Game` objects to represent those games.
- For each goal in a game, instantiate a `Goal` object and add it to the `Goal` array in the appropriate `Game` object.

In the `main` method of `League`, you now create two teams, each with two players. To help remember which player plays for which team, player names and their team names start with the same letter. For example **G**eorge Eliot plays for The **G**reens; **R**obert Service plays for the **R**eds.

- a. At the top of the `main` method (below where it says `TODO code application logic here`), declare and instantiate a `Player` object.

```
Player player1 = new Player();
```

- b. Set the `playerName` attribute of the `Player` object to "George Eliot".

```
player1.playerName = "George Eliot";
```

- c. Declare and instantiate a new `Player` object and set its `playerName` attribute to "Graham Greene".

```
Player player2 = new Player();
player2.playerName = "Graham Greene";
```

- d. Declare and instantiate a third `Player` object and set its `playerName` attribute to "Geoffrey Chaucer".

```
Player player3 = new Player();
player3.playerName = "Geoffrey Chaucer";
```

- e. Create a `Player` array called `thePlayers` that comprises the three `Player` objects that you just instantiated.

```
Player[] thePlayers = { player1, player2, player3 };
```

- f. Declare and instantiate a `Team` object.

```
Team team1 = new Team();
```

- g. Set the `teamName` attribute of the `Team` object to "The Greens".

```
team1.teamName = "The Greens";
```

- h. Set the `playerArray` attribute of the `Team` object to the `Player` array `thePlayers`.

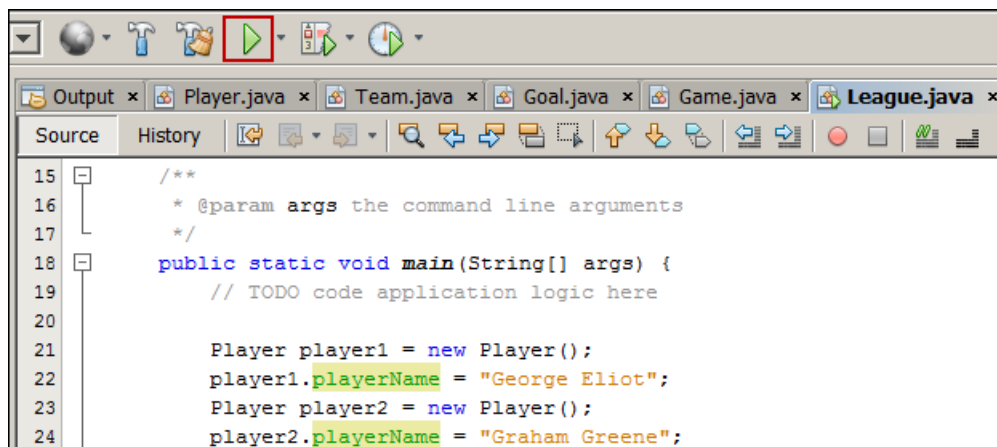
```
team1.playerArray = thePlayers;
```

9. Print out the players in the team "The Greens".

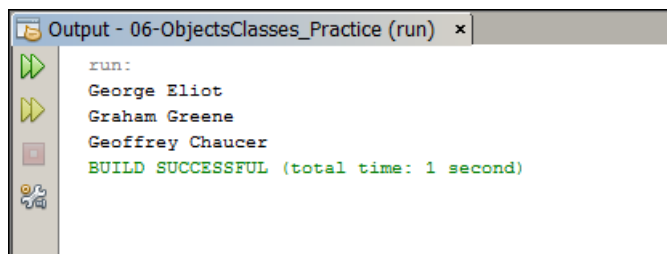
- a. Create a **for** loop that iterates through the array in `team1`.

```
for (Player thePlayer: team1.playerArray) {  
    System.out.println(thePlayer.playerName);  
}
```

- b. Run the project by clicking the green arrow.



- c. The output should look like the following. Notice that the output begins with a line that indicates that the project was run (as opposed to debugged, which you will learn about later). Then comes the three lines output by `System.out.println`. Finally, there is a line indicating that the project built successfully.



From now on, in these instructions, the output will show just the lines output by the code that you write. Here is what that would look like for the project that you just ran:

```
George Eliot  
Graham Greene  
Geoffrey Chaucer
```

10. Just above the **for** loop, add a statement to set `player1.playerName` to "Robert Service".

```
player1.playerName = "Robert Service";
```

- a. Now run the class again. What do you expect the output to be?

```
Robert Service  
Graham Greene  
Geoffrey Chaucer
```

- b. Remove or comment out the line that changes the name to Robert Service.

Technical Note:

Think about what has happened in this step. In your code, there are two references to "Robert Service" (the `String` that used to be "Graham Greene"). One of the references is `player1.playerName`; the other is `Team.playerArray[0].playerName`. Because both point to the same `String` in memory, if you use either reference to replace the old `String` with a new `String`, both references will now point to this new `String`.

11. Now create a second team with the reference `team2`. However, this time don't use the temporary references `player1`, `player2`, `player3`. Instead, create the team, create a new `Player` array and assign it to the `playerArray` reference of the `Team` object that you just created.

- a. Just above the `for` loop block, create a `Team` called `team2` and name it "The Reds".

```
Team team2 = new Team();  
team2.teamName = "The Reds";
```

- b. Create a three-element `Player` array and assign it to the `playerArray` reference of the `Team` object that you just created.

```
team2.playerArray = new Player[3];
```

- c. Add a player named "Robert Service" to the first element of `playerArray`.

```
team2.playerArray[0] = new Player();  
team2.playerArray[0].playerName = "Robert Service";
```

- d. Add two new `Player` objects with `playerName` attributes set to "Robbie Burns" and "Rafael Sabatini" (or use names of your own choosing). Add these players to the second and third elements of the `playerArray` array. Copy and paste may save some time here.

```
team2.playerArray[1] = new Player();  
team2.playerArray[1].playerName = "Robbie Burns";  
team2.playerArray[2] = new Player();  
team2.playerArray[2].playerName = "Rafael Sabatini";
```

- e. Copy and paste the **for** loop so that the second loop now prints out the names of the players on team2 (for the second loop change the reference to the `Team` object to `team2`). Run the application. Your output should look like this.

```
George Eliot  
Graham Greene  
Geoffrey Chaucer  
Robert Service  
Robbie Burns  
Rafael Sabatini
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

Practice 6-2: Creating a Soccer Game

Overview

In this practice, you use the classes that you have created and populated to create a new `Game` object and add some goals.

Tasks

1. Open the project **06-ObjectsClasses_Practice2**.
2. Create a game and add some goals.
 - a. In the `League` class, delete the two **for** loops that print out the names of the players on `team1` and `team2`.
 - b. At the bottom of the method, create a `Game` and populate the `homeTeam` and `awayTeam` attributes.

```
Game currGame = new Game();
currGame.homeTeam = team1;
currGame.awayTeam = team2;
```

- c. Create a `Goal` object to give the home team a 1–0 lead.

```
Goal goal1 = new Goal();
goal1.thePlayer = currGame.homeTeam.playerArray[2];
goal1.theTeam = currGame.homeTeam;
goal1.theTime = 55;
```

- d. Put this `Goal` object in a `Goal` array, and then assign this `Goal` array to the `goals` attribute of the `Game` object.

```
Goal[] theGoals = {goal1};
currGame.goals = theGoals;
```

- e. Print out the score of the game (if there was more than one goal, you would need to use a loop)

```
System.out.println("Goal scored after " +
currGame.goals[0].theTime + " mins by " +
currGame.goals[0].thePlayer.playerName + " of " +
currGame.goals[0].theTeam.teamName);
```

- f. When you run the code you should see this output.

```
Goal scored after 55.0 mins by Geoffrey Chaucer of the Greens
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

**Practices for Lesson 7:
Manipulating and Formatting
the Data in Your Program**

Practices for Lesson 7: Overview

Practices Overview

In these practices, you will use Javadocs to find useful methods for String manipulation and then work with these methods.

For Instructor Use Only.
This document should not be distributed.

Practice 7-1: Manipulating Text

Overview

In this practice, you write code to search for a particular player and print out player details. You also reverse player names so that the family name is printed first.

Tasks

1. Open the **07-ManipulateFormat_Practice1** project in NetBeans.
2. Suppose you want to find out whether the Blues have a player called Sabatini. Or is the name Sabatine or Sabadini? Look in the Javadocs for the `String` class to find a method that you can use to find a region of text.
 - a. Open the Javadocs and go to the `String` class. You can do this more easily by clicking `java.lang` to select the package, and then clicking `String` in the list of classes that are shown as being in `java.lang`.
 - b. You are looking for a method that tells you whether a `playerName String` contains a particular string. The first method that looks promising is `contains`. What does it return? A `boolean` type. That is what you want; you could write a simple *if* statement to do something when a particular family name is found.
 - c. Now look to see what parameter it takes. It takes a `CharSequence` type. This is a special type called an *interface*. This course has yet not covered interfaces, but they will be covered in the “Using Interfaces” lesson. So keep looking for another suitable method.
 - d. The next suitable method is possibly `endsWith`. This could work if you are looking for family name and are sure of the spelling, but may be less suitable if you are not sure of the spelling and only know a region in the middle of the name.
 - e. The next suitable method is `matches`. It returns a `boolean`, so that is good. Look to see what parameters it requires. It takes a `String`, a type you already know about.
 - f. Click the `matches` link for further information.
 - g. You can see here that the `String` that you pass into `matches` should be something called a *regular expression*. You may already know what this is, but if you do not, just click the regular expression link.
 - h. This is a slightly daunting screen, because regular expressions are very powerful. However, you need to do a very simple match. Assume that you are looking for a player with a family name that starts with the letters “Sab”. This will be after the first name but not stretch all the way to the end of the string. Therefore, the regular expression will be this exact string plus some additional special characters to “skip over” the first name and then to “skip over” the end of the string. A single dot matches any character, but you do not know how many you need. But if you follow a single dot with a `*`, this will give you the regular expression that you need—`.*Sab.*`

3. Write code to find the player in `team2` whose name contains the string "Sab".
- In the `League` class, add code for a simple loop to iterate through all the players who play for `team2`. Use `System.out.println` to output the player name. (You may be able to copy and paste one of the **for** loops that you created previously). Put your code at the bottom of the `main` method.

```
for (Player thePlayer: team2.playerArray) {  
    System.out.println( thePlayer.playerName);  
}
```

- Add an **if** statement below the `System.out.println` statement to test whether a match is found for "Sab". Use the regular expression from step 1h. Move the `System.out.println` statement inside the **if** clause so that you can tell whether the name "Sab" has been found.

```
for (Player thePlayer: team2.playerArray) {  
    if (thePlayer.playerName.matches(".*Sab.*")) {  
        System.out.println("Found " + thePlayer.playerName);  
    }  
}
```

- Run the application. The output should be:

```
Goal scored after 55.0 mins by Geoffrey Chaucer of The Greens  
Found Rafael Sabatini
```

4. Assuming that the name is found, print out just the last name of the player that has been found.

Technical discussion:

In an exercise during the lesson, you used the `indexOf` and `substring` methods to return part of a name. You could do that again, but perhaps it might be worth first checking whether there is a single method that can accomplish this. What about `split`? It returns a `String` array that is split around a regular expression. Because there is a space between a player's first and second names, you can split around a space and then the second index of the array returned will be the player's last name.

- Within the **if** block, add the following line of code to split the `String` around a space and return a `String` array:
- You could now assign this to a newly declared array. However, why not just modify this statement so that it returns the second element of the array? All you need to do is add the square brackets (with the index) at the end. Do this now. You will see an error but do not worry about it.

```
thePlayer.playerName.split(" ");
```

```
thePlayer.playerName.split(" ")[1];
```

- Pass the above statement into a `System.out.println` method to print out the second name.

```
System.out.println("Last name is " +  
thePlayer.playerName.split(" ")[1]);
```

- d. Test the code.

```
Goal scored after 55.0 mins by Geoffrey Chaucer of The Greens  
Found Rafael Sabatini  
Last name is Sabatini
```

5. Display all players of `team1` in the format `lastname, firstname`. Again, you can use the `split` method to split `playerName` into two separate `Strings`.

- a. Create a loop that iterates through the players in `team1`. Inside a loop, split the `String` into two separate `Strings` in a `String` array.

```
for (Player thePlayer: team1.playerArray) {  
    String name[] = thePlayer.playerName.split(" ");  
}
```

Technical discussion:

You use a `StringBuilder` to build up the name in the new format. Note that you could use `String` concatenation for this. While this may only create a few extra `String` objects, this could make quite a difference when reformatting a lot of `String` objects. Look at the Javadocs for `StringBuilder`.

- b. Just before the **for** loop, instantiate a new `StringBuilder` object. Note that if you instantiate the `StringBuilder` within the loop, it will be instantiated on every iteration of the loop!

```
StringBuilder familyNameFirst = new StringBuilder();
```

- c. Within the loop (after the `String` has been split), use the `append` method to add the second element of the name array because that is the player's family name.

```
familyNameFirst.append(name[1]);
```

- d. Use the `append` method to add a comma and a space.

```
familyNameFirst.append(", ");
```

- e. Use the `append` method to add the player's first name.

```
familyNameFirst.append(name[0]);
```

- f. Print out the full family name. Note that you can pass a `StringBuilder` into the `System.out.println` method.

```
System.out.println(familyNameFirst);
```

The output will look like this (output new to this step is **bolded**).

```
Goal scored after 55.0 mins by Geoffrey Chaucer of The Greens  
Found Rafael Sabatini  
Last name is Sabatini  
Eliot, George  
Eliot, GeorgeGreene, Graham  
Eliot, GeorgeGreene, GrahamChaucer, Geoffrey
```

- g. This is not quite right. You need to empty the `StringBuilder` each time after `System.out.println` is invoked. Look at the Javadocs to find a suitable method. You can use the `delete` method, but it requires an index to the start and end of the String in the `StringBuilder`. Fortunately, the beginning index is zero, and the end index can be found with the `length` method. Add the following code immediately below the `System.out.println` statement.

```
familyNameFirst.delete(0, familyNameFirst.length());
```

- h. Run the project again. This time you should get a list of the names of the players in `team1` with the family name listed first. (Output new to this step is bolded.)

```
Goal scored after 55.0 mins by Geoffrey Chaucer of The Greens
Found Rafael Sabatini
Last name is Sabatini
Eliot, George
Greene, Graham
Chaucer, Geoffrey
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

**Practices for Lesson 8:
Creating and Using Methods**

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 8: Overview

Practices Overview

In these practices, you will create and use methods.

For Instructor Use Only.
This document should not be distributed.

Practice 8-1: Using Methods

Overview

In this practice, you tidy up your code so that it uses methods. You will create individual methods `createTeams`, `createGames`, `playGames`, and `showResults`. Eventually, the `main` method will only have calls to these individual methods.

Tasks

1. Open the **08-Methods_Practice1** project.
2. Create a `createTeams` static method. It will require no parameters and return a `Team` array. Think about what it means for this to be a static method. You will revisit this question later.

- a. Go down to the end of the `main` method of `League`. After the closing brace of the `main` method, add a new static method, `createTeams`, that returns a `Team` array. Notice that NetBeans temporarily shows an error because the new method declares that it returns an array of type `Team` but there is no code to do that yet.

```
public static Team[] createTeams() {  
}
```

- b. Find the existing code in `League` that creates the `Team` array (there are comments at the beginning and end of this block of code to help you find it).
- c. Highlight this code, and then cut (not copy) and paste it into the `createTeams` method.
- d. At the end of `createTeams`, add a line to put the two `Team` references into a `Team` array, and then add a return statement to return this array.

```
Team[] theTeams = {team1, team2};  
return theTeams
```

3. Immediately below the `createTeams` method, create another static method, `createGames`. You will pass in an array of type `Team`, and receive back an array of type `Game`.
 - a. Create a new method, `createGames`, just below `createTeams`. This too will show an error until later when you add the return statement.

```
public static Game[] createGames(Team[] theTeams) {  
}
```

- b. Now add code to create just one game (so you will be returning an array with one element).

```
Game theGame = new Game();
```

- c. This `Game` object needs to have `homeTeam` and `awayTeam` set; therefore, do this now using two teams from the `Team` array that has been passed into the method.

```
theGame.homeTeam = theTeams[0];  
theGame.awayTeam = theTeams[1];
```

- d. Create a `Game` array with `theGame` as its only element, and return it.

```
Game[] theGames = {theGame};  
return theGames;
```

4. Add code to call these new methods.

- a. At the beginning of the `main` method, add a call to the `createTeams` method.

```
Team[] theTeams = createTeams();
```

- b. Immediately after the call to the `createTeams` method, add a call to the `createGames` method.

```
Game[] theGames = createGames(theTeams);
```

- c. Now you have an array of type `Game` (albeit with just one element), but the remaining code in the `main` method works with a single `Game`, `currGame`. Modify the line that currently creates a `Game` by instantiating a new `Game` object so that instead it now gets the `Game` object from the `currGame` array.

```
Game currGame = theGames[0];
```

- d. Delete these three lines from the `main` method of `League`.

```
Game currGame = new Game();  
currGame.homeTeam = team1;  
currGame.awayTeam = team2;
```

(These are the lines that create a new `Game` and assigned teams to `currGame.homeTeam` and `currGame.awayTeam`. They are no longer needed, because that is now done in the `createTeams` method. This should clear all the NetBeans syntax errors.)

- e. Run the `main` method. The lines of code that creates the goals and reports the result should now give you the following output:

```
Goal scored after 55.0 mins by Rafael Sabatini of the Reds
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

Practice 8-2: Creating Game Data Randomly

Overview

In this practice, you use a provided utility class to create code that will create `Goal` objects with randomly set attributes. This will help with testing, allowing multiple games to be played, and so model a real league.

Tasks

1. Open the **08-Methods_Practice2** project.
2. The code that currently creates a `Goal` is hard-coded (meaning it is the same every time the method runs). Replace this code with a call to a method of a utility class that returns a played game. You will write the code to determine how many goals, and the utility method will randomly determine which player scores each goal.

Technical discussion:

You can use the `random` method of the `Math` class to determine how many elements to put in a `Goal` array that will represent the goals scored in the game. If you look up the `Math` class in the Java documentation, you will see that it has a `random` method. This method returns a `double` that is greater than or equal to 0.0 and less than 1.0. But you will need to turn this into an `int` that you can use to create the `Goal` array.

Suppose you decide that there should be a maximum of six goals. To randomly decide how many goals should be in a particular game, you can use `Math.random()`.

`Math.random() * 7` will produce a `double` that is greater than or equal to 0.0 and less than 7.0. Now, how can you turn this result into an `int` that is 0, 1, 2, 3, 4, 5, or 6?

How about `Math.round()`? That looks promising; however, it returns a `long` and not an `int`, and you need an `int` to initialize the array. You would have to cast the returned value to an `int`. But wait, if you are casting to an `int` anyway, there should be no need to use `Math.round()`; the cast will ensure that the result is an `int`.

- a. Delete the current code that creates a `Goal` and assigns it to a `Team` and a `Game`.
- b. Comment out (do not delete) the `System.out.println` statement that prints out the goal details.
- c. Just after the line that assigns `currGame`, add the code to create a random number of goals between 0 and 6. Also add `System.out.println` to print out the result. Note the parentheses around `Math.random() * 7`. Without these, the cast would operate on the `double` returned by `Math.random()` and, therefore, `numberOfGoals` would always be zero.

```
int numberOfGoals = (int)(Math.random() * 7);  
System.out.println(numberOfGoals);
```

- d. Run the project a few times to make sure that `numberOfGoals` is a random number between 0 and 6.

3. Now create the `Goal` array.

- a. Below the code that you just added, create a `Goal` array using `numberOfGoals` to give it the correct number of elements. Note that it is not incorrect code to create an array with zero elements! Put your line of code before the `System.out.println` statement and modify `System.out.println` to print out the length of the array.

```
Goal[] theGoals = new Goal[numberOfGoals];  
System.out.println(theGoals.length);
```

- b. Run the project a few times to make sure that it is working correctly and that the array length is randomly between 0 and 6.

Technical discussion:

Now that you have a `Goal` array, you can use the special utility class to populate the `Goal` array with `Goal` objects (or strictly speaking with references to `Goal` objects). Each goal will be randomly created with a random time, one or the other of the teams, and a player. The utility class that you need is called `GameUtils` and it is in the `utility` package. It has a static method called `addGameGoals` that takes a `Game` as a parameter.

- c. Delete the `System.out.println` statement that prints the length of the `Goal` array and, in its place, add code to assign the `Goals` array to the `Game` object created earlier.

```
currGame.goals = theGoals;
```

- d. Now call the `addGameGoals` method of the `GameUtils` class.

```
GameUtils.addGameGoals(currGame);
```

NetBeans will show an error. Click the red question mark in the left margin, and select `add import for utility.GameUtils`.

- e. Run the application. It should run without error, but there will not be any output.

4. Add another for loop that reports on the goals in the current game.

- a. Just below the call to the `addGameGoals` method add a **for** loop to iterate through each of the goals of the current game.

```
for (Goal currGoal: currGame.goals) {  
}
```

- b. Now add a `System.out.println` statement to list each goal, when it was scored, by which team, and by which player. Something similar to the following.

```
System.out.println("Goal scored after " +  
    currGoal.getTime() + " mins by " +  
    currGoal.getPlayer().getPlayerName() +  
    " of " + currGoal.getTeam().getTeamName());
```

- c. Run the code. Here is some typical output, but the output will be different each time the application is run:

```
Goal scored after 29.0 mins by Graham Greene of The Greens
Goal scored after 36.0 mins by Robbie Burns of The Reds
Goal scored after 47.0 mins by Robbie Burns of The Reds
Goal scored after 77.0 mins by Geoffrey Chaucer of The Greens
```

Creating a `playGame` Method

5. Examine the code that you created for creating a random result for a game. Even though it is only four lines, it seems like a block of code that should be in its own method. But where should the method go? It could, of course, be in the `League` class. However, the data it needs is a `Game` object and its `goals` attribute, so it makes sense that it should be in the `Game` class.

- a. Go to the `Game` class and create a new method `playGame`. Does it need any parameters passed in? Or returned? It would seem not, because it can get its data from the `Game` object itself and modify the `Game` object. However, it should not be static. Why not?

```
public void playGame() {
}
```

- b. Now cut the code that plays a game from `League`, and paste it into the `playGame` method of `Game`. That will be the four lines starting just after:

```
Game currGame = theGames[0];
```

to just above the loop that prints out the game result (including the line below).

```
GameUtils.addGameGoals(currGame);
```

As you paste it into `Game`, NetBeans will recognize that you need an import for `GameUtils`. Click OK in the Import Classes dialog box.

- c. Does it need any modifications to run? Do you see any errors in the code in NetBeans? NetBeans indicates that it cannot find the variable `currGame`. Should you pass it into the method after all? No, because you are actually in the `Game` class! Therefore, just replace `currGame` with `this`, and the code should be fine.
- d. Back in `League`, just above the loop to print the result of the game, add a call to the `playGame` method. Note that `currGame` is already there and points to the `Game` object; therefore, all you need is:
- ```
currGame.playGame();
```
- e. Run the code. It should work exactly as before.

6. Move the code that describes the game into its own method. Where should that go? Again, look at the data it requires, data that is held on the `Game` object.

- a. Create a new method on the `Game` class to return a description of the game. This will need to return a `String`. (When you create the method, you will see a NetBeans error until later when you add the line to return a `String`).

```
public String getDescription() {
}
```

- b. Now cut the **for** loop from `League` and paste it in the `getDescription` method. Your code will look like this (there will be NetBeans errors):

```
for (Goal currGoal: currGame.goals) {
 System.out.println("Goal scored after " +
 currGoal.getTime() + " mins by " +
 currGoal.getPlayer().playerName +
 " of " + currGoal.getTeam().teamName);
}
```

- c. Notice the error like the one you had before with the `currGame` variable not being found? As before, replace `currGame` with `this`.
- d. Look at the remaining error: no return statement. The method needs to return a `String`. What `String` should be returned? A `String` that is made up of each line that is currently passed into the `System.out.println` method. You can use a `StringBuilder` and its `append` method for this. Instantiate a `StringBuilder` before the start of the loop.

```
StringBuilder returnString = new StringBuilder();
```

- e. Previously, each iteration of the loop passed a `String` to the `System.out.println` method to print. Instead, this now needs to be appended to the `StringBuilder`. Replace `System.out.println` with a `returnString.append` method—like this:

```
returnString.append ("Goal scored after " +
 currGoal.getTime() + " mins by " +
 currGoal.getPlayer().playerName +
 " of " + currGoal.getTeam().teamName);
```

- f. Add a return after the loop to return the entire `StringBuilder`. Notice that because a `StringBuilder` is not a `String` and, therefore, not the correct type, you must call the `toString` method on `returnString`.

```
return returnString.toString();
```

- g. Back in the `League` class, at the bottom of the `main` method, add a `System.out.println` statement to print out the game description.

```
System.out.println(currGame.getDescription());
```

Can you see what is happening here? The statement `currGame.getDescription()` is being called and returns a `String` to the `System.out.println` method, which prints it to the console.

- h. Run the code a few times to see whether it is working correctly. What did you see? `System.out.println` puts each line of text about each goal on a new line but the `append` method did not add any newlines. Go back to the `getDescription` method and add `"\n"` at the end of the line appended to `returnString`.

```
returnString.append ("Goal scored after " +
 currGoal.getTime() + " mins by " +
 currGoal.getPlayer().playerName +
 " of " + currGoal.getTeam().teamName +
 "\n");
```

- i. Run the code again. Each goal should now be described on a new line.

You may notice that the `getDescription` method could use the `append` method more extensively by replacing every concatenation operator with an `append` method. For now, though, it is probably more readable as is, and you will make changes to this method later in the course.

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

## Practice 8-3: Creating Overloaded Methods

---

### Overview

In this practice, you create an overloaded method. At the moment the `playGame` method automatically creates a random set of goals for a game with a maximum of 6. This may be a good default number, but it might be useful to call the `playGame` method and pass in the maximum number of goals. To allow for either accepting the default (maximum six goals), or specifying a different maximum, you need to overload the `playGame` method in `Game`. This means that there will be two methods, one that receives no parameters, and one that receives the parameter specifying the maximum number of goals.

### Tasks

1. Open the **08-Methods\_Practice3** project.
2. The first step is to modify the current `playGame` method of `Game` to work with an `int` parameter passed in as `maxGoals`. Why not create a brand new version of the `playGame` method? The reason should become apparent as you complete this section.
  - a. In `League`, change the line that calls the `playGame` method of `Game` so that it passes an `int` of value 3. The line will now look like this (note that this will temporarily show as an error because no method with this signature currently exists).

```
currGame.playGame(3);
```
  - b. Go to the `playGame` method of `Game` and change its signature to:

```
void playGame(int maxGoals)
```
  - c. Modify the line in the `playGame` method that determines the number of goals in the game by replacing the literal 7 with `(maxGoals + 1)`—and do not forget the parentheses!

```
int numberOfGoals = (int)(Math.random() * (maxGoals + 1));
```

Why + 1? Remember that casting to an `int` will round down to the nearest `int` value.
  - d. This should fix the error in `League`. Run the application a few times. It should run as before but after a few tries you should see that it now never goes over 3 goals per game.
3. Now add a no parameter `playGame` method.
  - a. Add another `playGame` method to `Game`. It should return nothing and receive no parameters.

```
public void playGame() {
}
```
  - b. You now have a `playGame` method that is overloaded. However, the no parameter version does not do anything yet. How to code this method? Simply call the parameter version of the `playGame` method with your chosen default `maxGoals`.
  - c. Add this line to the no parameter `playGame` method:

```
playGame(6);
```



- d. Go back to `League` and change the line that calls the `playGame` method of `Game` so that it passes no parameters.
- e. Run the application a few times. You should now see that the maximum number of goals is now 6.
- f. Why do you think it is better to have the no parameter version of the `playGame` method call the parameterized version of the `playGame` method? The answer is code duplication. As far as possible you do not want to have two pieces of code do the same job. Later if a bug is discovered in the `playGame` method, there will be only one `playGame` method to fix.

## Making `League` an Instance

In this section, you will instantiate `League` and run the various methods as instance methods. This would allow `League` to represent more than one tournament or league.

- 4. Modify `League` so that it is instantiated and the methods are called on the instantiated `League` object.
  - a. For the method `createGames`, remove the static keyword. The `createGames` method now becomes an instance method, while the `createTeams` method remains a static method.

### Technical discussion:

At this point, you will see an error in the `main` method. If you hover the mouse over the error, you can see what it is. The non-static method is `createGames`. What is meant by the term static context? Because the `main` method is a static method, it is running on the `League` class, NOT on a `League` object. Therefore, a call to the `createGames` method is interpreted as a call to a static method, `createGames`. If instead you specify a reference to a `League` object, the method call will work.

- b. Just below the `main` method signature, instantiate a new `League` object with a reference, `theLeague`.

```
League theLeague = new League();
```
- c. Modify the call to the `createGames` method so that it calls the `createGames` method on this new object.

```
Game[] theGames = theLeague.createGames(theTeams);
```
- d. Run the application. It should work fine even though the `createTeams` method is still a static method. This is not a recommended approach. It is just to illustrate that the static method `main` can call other static methods such as `createTeams`; however, to access a non-static method, the reference must precede the method name (using the dot notation).
- e. Make `createTeams` an instance method also, and modify the call to it so that it works in the same way as the `createGames` method.

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.



## **Practices for Lesson 9: Using Encapsulation**

For Instructor Use Only.  
This document should not be distributed.

## Practices for Lesson 9: Overview

---

### Practices Overview

In these practices, you will modify all classes to encapsulate all the attributes and add constructors.

For Instructor Use Only.  
This document should not be distributed.

## Practice 9-1: Encapsulating Attributes

---

### Overview

In this practice, you use the NetBeans refactor feature to encapsulate all the classes in your application.

### Tasks

1. Open the **09-EncapConstructors-Practice1** project.
2. Encapsulate the `Player` class and ensure that the code in `League` now accesses the encapsulated class correctly.
  - a. Right-click `Player` and select Refactor > Encapsulate Fields.
  - b. In the Encapsulate Fields dialog box, select `playerName` as the field to encapsulate (it will be selected by default), and check the boxes to create the `getPlayerName` method and the `setPlayerName` method. The options should be as follows (these are the defaults).

| Option                | Choices or Values       |
|-----------------------|-------------------------|
| Insert Point          | Default                 |
| Sort By               | Getter/Setter pairs     |
| Javadoc               | Create default comments |
| Fields Visibility     | private                 |
| Accessor's Visibility | public                  |

- c. Click Refactor (accepting the other options).
- d. Run the project a few times. It should run as before.

#### Technical discussion:

Look at the code of the `Player` class. You should see that the `playerName` attribute is now marked private, and that two new methods `getPlayerName` and `setPlayerName` have been added. But if the `playerName` attribute is now private (unable to be accessed other than within the `Player` class), how will the `createTeams` method of `League` function?

Examine the `createTeams` method of `League`. You will see that all the code in this method that previously accessed `playerName` of `Player` directly, now does so using the `setPlayerName` method. NetBeans is capable of not only modifying `Player` to encapsulate its fields, but also of modifying any code that uses `Player` to ensure that it now complies with the new encapsulated version of `Player`.

3. Use NetBeans' refactor feature to encapsulate the remaining classes; `Team`, `Game` and `Goal`.
  - a. As you did for the `Player` class, encapsulate the `Team` class. Encapsulate both fields, creating setter and getter methods.

- b. Examine the `Team` class to see that the fields have been made private and that the getter and setter methods have been created.

Test that the application is running as before.

- c. Encapsulate the `Game` class. Encapsulate all three fields, creating setter and getter methods (you can use Select All for this).
- d. Click **Save All** to save your changes.
- e. Examine the changes that have taken place and ensure the application still runs correctly.
- f. Encapsulate the `Goal` class. Encapsulate all three fields, creating setter and getter methods (you can use Select All for this).
- g. Examine the changes that have taken place and ensure the application still runs correctly.
- h. Example Output:

```
Goal scored after 27.0 mins by Robbie Burns of The Reds
Goal scored after 79.0 mins by George Eliot of The Greens
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

## Practice 9-2: Adding Constructors

---

### Overview

In this practice, you add constructors to the classes so that objects can be instantiated and have their data populated in a single line.

### Tasks

1. Open the **09-EncapConstructors-Practice2** project.
2. Add a constructor to the `Player` class that accepts a `String` parameter for the player's name.
  - a. After the `playerName` attribute, add a constructor that returns void and accepts a `String` parameter name. (You will see an error appear in `League`. Ignore it for the now.)

```
public Player(String playerName) {
}
```

- b. Add a line to this constructor method that sets the `playerName` attribute to the `String` that has been passed in. Notice that the parameter name passed in is the same as the attribute name, but go ahead and add the following code anyway. (This will cause a problem later but leave it like this for now so you will see what can happen.)  
`playerName = playerName;`
3. Fix the errors that appear in the `League` class. (The fix might not be to the `League` class.)
    - a. Examine the `createTeams` method. You will see some errors. Why does the code no longer work? After all, you did not remove any code from `Player`; you only added a constructor that takes a `String` parameter. If you look more closely, you will see that it is the statement that creates a new `Player` that is now the problem.

#### Technical discussion:

Because there was no explicit constructor in the `Player` class previously, Java assumed that a default no-argument constructor was required, and the `Player` class behaved as if it were there. That is why previously the new keyword worked correctly when no arguments were passed.

However, now that you have added a constructor with a parameter, Java no longer assumes that you require a no-argument default constructor. Therefore, all the code that requires a no-argument constructor fails.

- b. Below the constructor that you just added, add an explicit no-argument constructor to the `Player` class. You can do this in one line.  
`public Player() { }`
- c. Click **Save All**.
- d. Examine the `createTeams` method of `League`. You will see that there are now no problems and that the application runs correctly.

4. Modify the `League` class to use the `Player` constructor.

- a. Modify the first line of the `createTeams` method to work with the new parameterized constructor of `Player`. The current line is:

```
Player player1 = new Player();
```

Change it to:

```
Player player1 = new Player("George Eliot");
```

- b. Delete the next line. It should no longer be necessary to explicitly set the player name.
- c. Run the application a few times and look out for George Eliot to ensure that the player name is being set correctly.
- d. Did you see George Eliot as a scorer any time? No? How about null—did the player name come up as null at all? If so, you should suspect that there is something wrong with the constructor you created.
- e. Go to the constructor in `Player`. Notice that NetBeans has marked the line that sets `playerName` with a warning. Hover over the line to see what NetBeans reports. The value is never used, because it is assigned to itself. Any use of `playerName` within the constructor method means the local variable of the constructor method. Therefore, the `playerName` reference passed into the constructor is assigned to itself. How can you instead assign this value to the attribute `playerName` of the object currently being executed?

You need a reference to this object—remember what the keyword is? It is `this`. So the following will work correctly. Make this modification now.

```
this.playerName = playerName;
```

**Technical discussion:**

**this** is a reference to the `Player` object and, therefore, `this.playerName` refers to the `playerName` attribute of the object created by this constructor. `playerName` refers to the local variable `playerName` passed into the method. Only within the method does `playerName` refer to the local variable. Outside the method (and, therefore, outside the scope of the local variable `playerName`), the name `playerName` reverts to the attribute of the object.

You can see this clearly by looking at the (automatically created) getter and setter methods. In the `setPlayerName` method, where a parameter `playerName` is passed in, `playerName` becomes a local variable and, therefore, the attribute `playerName` must be qualified with the reference `this`. However, in the `getPlayerName` method, `playerName` refers to the object attribute `playerName`.

Note that it is also possible to get around this scoping problem by ensuring that the name of the parameter in the method signature is different than the attribute name.

- f. Modify the remainder of the `createTeams` method so that all `Player` objects are instantiated with the player name and remove all the (now unnecessary) `setPlayerName` methods. Notice how the code is shortened and simplified.
- g. Test to make sure that everything still works correctly.



5. Add two constructors to the `Team` class—the first for setting the team name and the second to pass in the array of players.

- a. Open the `Team` class and add a constructor that sets the `Team` name (just as you did for `Player`).

```
public Team(String teamName) {
 this.teamName = teamName;
}
```

- b. Below this constructor, add another constructor that receives the `teamName` and an array of `Player` objects.

```
public Team (String teamName, Player[] players) {
}
```

- c. Add a line to set the attribute `playerArray` to the `Player` array that has been passed in. Note that the `this` keyword is not necessary in this case, because the names are different, but it aids readability.

```
public Team (String teamName, Player[] players) {
 this.playerArray = players;
}
```

- d. Add another line to set the `teamName` attribute—but wait, could you just call the previously written constructor? Yes! Again, this helps ensure that there is no code duplication and is common in constructors. Notice how the call is made—again by using `this`. Also notice that it must be the first line of the constructor.

```
public Team (String teamName, Player[] players) {
 this(teamName);
 this.playerArray = players;
}
```

- e. Add a default no-argument constructor below this constructor.
- f. Modify the `createTeams` method of `League` so that, for the first team (`team1`), the team name and the `Player` array are passed into the `Team` constructor. Currently the code looks like this:

```
Team team1 = new Team()
```

Modify this line to.

```
Team team1 = new Team("The Greens", thePlayers);
```

- g. Remove the following two lines, because they are no longer necessary.

```
team1.setTeamName("The Greens");
team1.setPlayerArray(thePlayers);
```

- h. `team2` is created a little differently, but, because there is a no-argument constructor in `Team`, you can leave it as is.
- i. Test the application to ensure that it still works as before. Notice how much neater the code for creating `team1` now is.

6. Add constructors for the `Game` class.
  - a. Add a constructor that receives two `Team` parameters.

```
public Game (Team homeTeam, Team awayTeam) {
 this.homeTeam = homeTeam;
 this.awayTeam = awayTeam;
}
```

- b. Modify the code in the `createGames` method of `League` to use this constructor. The entire method will now be much shorter.

```
public Game[] createGames(Team[] theTeams) {
 Game theGame = new Game(theTeams[0], theTeams[1]);
 Game[] theGames = {theGame};
 return theGames;
}
```

- c. Run the application to ensure that it still works properly.
7. Modify the `createGames` method to add some more games to the `Game` array so that four games are played in total. Because there are only two teams, the teams will play each other four times.
  - a. Copy the line of code that instantiates a new `Game`. Paste it immediately below. You will see errors because `theGame` is now repeated.
  - b. Correct the errors by using the name `theGame2` in the new line of code. In addition, reverse the order of the parameters to reverse which team is home and which one is away.
  - c. Add the new `Game` to the `Game` array. The entire `createGames` method will now look like this (with the new lines and modifications shown in bold):

```
public Game[] createGames(Team[] theTeams) {
 Game theGame = new Game(theTeams[0], theTeams[1]);
 Game theGame2 = new Game(theTeams[1], theTeams[0]);

 Game[] theGames = {theGame, theGame2};
 return theGames;
}
```

- d. Copy and paste the two lines that instantiate `Game` objects and modify the names to `theGame3` and `theGame4`. Add these new games to the `Game` array. The method now looks like this.

```
public Game[] createGames(Team[] theTeams) {
 Game theGame = new Game(theTeams[0], theTeams[1]);
 Game theGame2 = new Game(theTeams[1], theTeams[0]);
 Game theGame3 = new Game(theTeams[0], theTeams[1]);
 Game theGame4 = new Game(theTeams[1], theTeams[0]);
 Game[] theGames = {theGame, theGame2, theGame3, theGame4};
 return theGames;
}
```

Note that repeating the code like this is not good practice; it would make much more sense to use a loop, and you will do that in Practice 11-1.

8. Modify the main method of `League` to play all games.

- a. Remove the line that currently sets up `currGame` and add a **for** loop around `currGame.playGame()` and the `System.out.println` statement so that all games are now played. It will look like this:

```
for (Game currGame: theGames) {
 currGame.playGame();
 System.out.println(currGame.getDescription());
}
```

- b. Run the application a few times. You will see something like the following. Each block of code represents a different game. (What do you think it means if there are less than four blocks? The answer is in the next practice!)

```
Goal scored after 29.0 mins by Graham Greene of The Greens
Goal scored after 36.0 mins by Robbie Burns of The Reds
Goal scored after 48.0 mins by Robbie Burns of The Reds
Goal scored after 77.0 mins by Geoffrey Chaucer of The Greens

Goal scored after 29.0 mins by Bertrand Russell of the Blues
Goal scored after 79.0 mins by Robert Service of the Reds

Goal scored after 19.0 mins by George Eliot of The Greens
Goal scored after 31.0 mins by Robert Service of The Reds
Goal scored after 56.0 mins by Rafael Sabatini of The Reds

Goal scored after 59.0 mins by Geoffrey Chaucer of The Greens
Goal scored after 77.0 mins by Robbie Burns of The Reds
```

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.



**Practices for Lesson 10:  
More on Conditionals**

## Practices for Lesson 10: Overview

---

### Practices Overview

In these practices, you will work with various combinations of ***if*** and ***else***, and you will also use the ***ternary*** operator. You will also use the debugger to see what is happening in an ***if-else*** clause.

For Instructor Use Only.  
This document should not be distributed.

## Practice 10-1: Using Conditionals

---

### Overview

In this practice, you enhance the `getDescription` method of the `Game` class in order to announce the name of the winning team. One way to do this is to iterate through the `Goal` array for the `Game`, incrementing `int` variables for either the home team or the away team.

### Tasks

1. Open the **10-Conditions\_Practice1** project.
2. Set up variables to hold the number of goals scored for each team.
  - a. At the start of the `getDescription` method in the `Game` class, declare two `int` variables to hold the score for each team. The variables should be called `homeTeamGoals` and `awayTeamGoals`.

```
int homeTeamGoals = 0;
int awayTeamGoals = 0
```

- - b. Inside the **for** loop that iterates through the `Goal` array, before the `returnString.append()` statement, write a test to determine which team scored each goal. Consider the `Team` in element zero of the `Team` array to be the `homeTeam`.

```
if (currGoal.getTheTeam() == homeTeam) {
 homeTeamGoals++;
}
else {
 awayTeamGoals++;
}
```

- - 
    - c. After the **for** loop, write another block that determines the following: if the game was a draw or, if there was a winner, who the winner was.

```
if (homeTeamGoals == awayTeamGoals) {
 ...// No code here yet;
}
else if (homeTeamGoals > awayTeamGoals) {
 ...// No code here yet;
}
else {
 ...// No code here yet;
}
```

- - 
    - 
    - d. Within each branch of the **if** statement, add appropriate text to the `String` `returnString`. The entire `if` block will now look something like this.

```
if (homeTeamGoals == awayTeamGoals) {
 returnString.append("It's a draw!");
}
```

```

else if (homeTeamGoals > awayTeamGoals) {
 returnString.append
 (returnString.append(homeTeam.getTeamName()+ " win"));
}
else {
 returnString.append
 (returnString.append(awayTeam.getTeamName() + " win"));
}

```

- e. Add another line after the **if-else** block to print the score.

```

returnString.append(" (" + homeTeamGoals + " - " +
 awayTeamGoals + ") \n");

```

- f. Immediately below the line that instantiates the `returnString` `StringBuilder` add a line that appends text to `returnString` to show which teams are playing.

```

returnString.append
 (homeTeam.getTeamName() +
 " vs. " + awayTeam.getTeamName() + "\n");

```

- g. Test the application. The lines below that are new output are bolded.

**The Greens vs. The Reds**

```

Goal scored after 17.0 mins by Rafael Sabatini of The Reds
Goal scored after 32.0 mins by Robert Service of The Reds
Goal scored after 35.0 mins by Geoffrey Chaucer of The Greens
The Reds win (1 - 2)

```

**The Reds vs. The Greens**

```

Goal scored after 21.0 mins by Rafael Sabatini of The Reds
Goal scored after 24.0 mins by George Eliot of The Greens
Goal scored after 29.0 mins by George Eliot of The Greens
The Greens win (1 - 2)

```

**The Greens vs. The Reds**

```

Goal scored after 36.0 mins by George Eliot of The Greens
The Greens win (1 - 0)

```

**The Reds vs. The Greens**

```

Goal scored after 4.0 mins by Geoffrey Chaucer of The Greens
Goal scored after 15.0 mins by Geoffrey Chaucer of The Greens
Goal scored after 43.0 mins by Geoffrey Chaucer of The Greens
Goal scored after 47.0 mins by Robert Service of The Reds
Goal scored after 82.0 mins by Geoffrey Chaucer of The Greens
The Greens win (1 - 4)

```

```

BUILD SUCCESSFUL (total time: 0 seconds)

```



## Determine the League Winner

### Technical discussion:

To determine the winner of the `League`, you need to award points to the winning team of each game. As you iterate through each game you will need to increment a variable for each team to add to this points score if they win. The obvious place for this variable is on the `Team` object and that is where you will store it. However, there are drawbacks to this approach. The number of points scored by a team can be derived from the data stored in the `Game` array, so having this stored separately on each `Team` object needs some care. What if the score of a game is changed after the points have already been awarded to the winner? Therefore, it is best to ensure that the number of points is calculated anew each time. There is also a strong case for storing the number of points a team scores on a separate class, perhaps called `TeamDisplay`; then you could ensure that only the `Team` class gets persisted to a database.

In this practice, you will use the approach that stores the points score on the `Team` object.

3. Add an `int` field on `Team` to store the number of points scored in the `League`.
  - a. Add the following `int` field to `Team`.

```
private int pointsTotal;
```
  - b. Use the refactor feature of NetBeans to create getter and setter methods for this field.
  - c. Add a further method that increments the `pointsTotal` field by the value passed in.

```
public void incPointsTotal(int pointsTotal){
 this.pointsTotal += pointsTotal;
}
```
4. Modify the `getDescription` method of `Game` to increment this field when you have determined a game winner.
  - a. In the **`if/else`** clause (toward the bottom of the `getDescription` method) award a team 2 points for a win, 1 point for a draw, and zero points for a loss. Here is what the **`if`** clause will look like now (new lines added are bolded).

```
if (homeTeamGoals == awayTeamGoals) {
 returnString.append("It's a draw!");
 homeTeam.incPointsTotal(1);
 awayTeam.incPointsTotal(1);
} else if (homeTeamGoals > awayTeamGoals) {
 returnString.append(homeTeam.getTeamName() + " win");
 homeTeam.incPointsTotal(2);
} else {
 returnString.append(awayTeam.getTeamName() + " win");
 awayTeam.incPointsTotal(2);
}
```

5. Create a new method in `League` to show the points scored by each team.

- a. Create a method called `showBestTeam` that receives a `Team` array and returns void.

```
public void showBestTeam(Team[] theTeams) {

}
```

- b. Within this method, add a `System.out.println` to print "Team Points" to the console.

```
System.out.println("\nTeam Points");
```

- c. Add a loop that iterates through the `Team` array and prints out the number of points that each team scores.

```
for (Team currTeam: theTeams){
 System.out.println(currTeam.getTeamName() + ":" +
 currTeam.getPointsTotal());
}
```

- d. At the bottom of the `main` method of `League`, add a call to the `showBestTeam` method.

```
theLeague.showBestTeam(theTeams);
```

- e. Try the application. The output will now show each team and the number of points they scored.

```
Team Points
The Greens: 2
The Reds: 4
```

6. Now add an ***if/else*** clause to determine which team actually won the league.

- a. Declare a variable to store the best team at the first line of the `showBestTeam` method. In the example code below, you arbitrarily set the `Team` at the first element of the `Team` array to be the `currBestTeam` initially. This will be replaced by a new best `Team` as the loop iterates through the `Team` array.

```
Team currBestTeam = theTeams[0];
```

- b. Within the loop, check whether `currTeam` has more points than `currBestTeam`. If it does, set `currBestTeam` to `currTeam`. Put this line below the `System.out.println` statement in the code. Use the ternary operator for this operation.

```
currBestTeam = currTeam.getPointsTotal() >
 currBestTeam.getPointsTotal() ? currTeam : currBestTeam;
```

- c. After the ***for*** loop, add a `System.out.println` to write out the name of the winning team to the console.

```
System.out.println("Winner of the league is " +
 currBestTeam.getTeamName());
```

- d. Run the application a few times to see whether the team with the most points gets reported as the best team. Does it work correctly?

- e. You should see that it works correctly as long as the two teams do not have the same number of points. In those cases, it simply reports the first one found. How could this be improved? One way would be to use the number of goals scored by a team to differentiate between two teams which are otherwise equal. You will explore this further in Practice 10-2.

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.

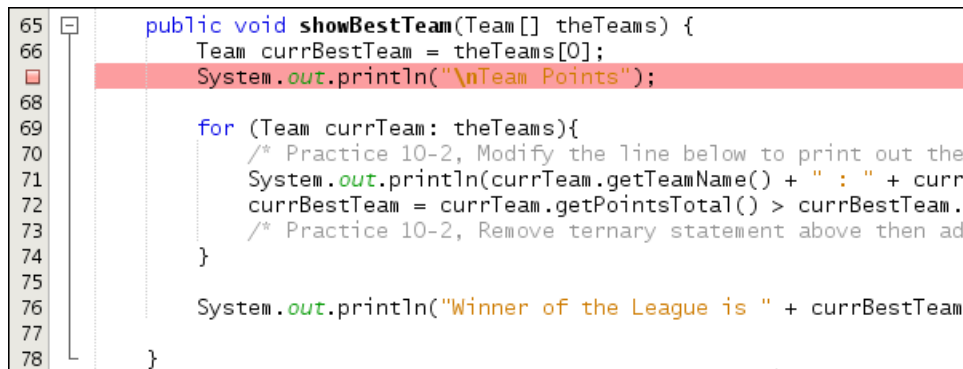
## Practice 10-2: Debugging

### Overview

In this practice, you will enhance the `showBestTeam` method to differentiate between teams that have the same number of points. Because this is some of the more complex code so far, it is a good time to look at some of the debugging features of NetBeans. In particular, you will step through the code line by line, seeing how each line modifies various attributes.

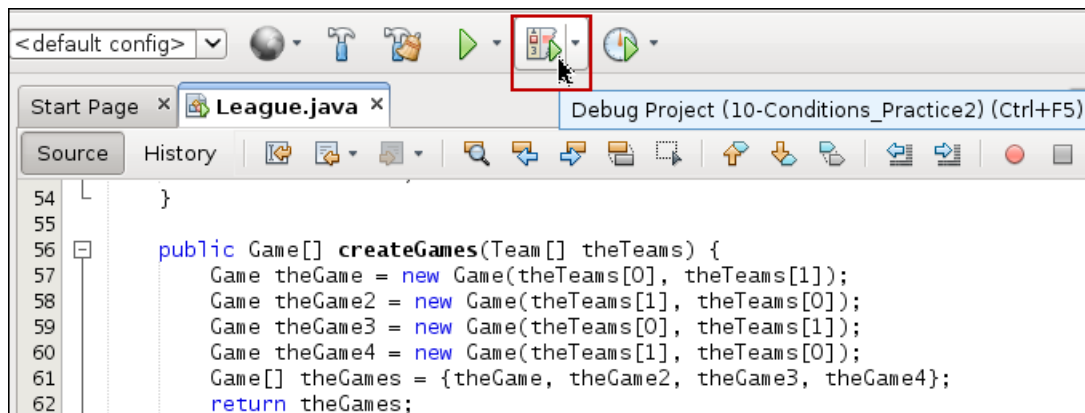
### Tasks

1. Open the **10-Conditions\_Practice2** project.
2. Use the debugger to follow the execution of the `showBestTeam` method.
  - a. Set a breakpoint in the `showBestTeam` method (in `League`). You can do this by clicking any of the line numbers. Set the breakpoint at the `System.out.println` just before the **for** loop. The breakpoint will look like this.



```
65 public void showBestTeam(Team[] theTeams) {
66 Team currBestTeam = theTeams[0];
67 System.out.println("\nTeam Points");
68
69 for (Team currTeam: theTeams){
70 /* Practice 10-2, Modify the line below to print out the
71 System.out.println(currTeam.getTeamName() + " : " + curr
72 currBestTeam = currTeam.getPointsTotal() > currBestTeam.
73 /* Practice 10-2, Remove ternary statement above then ad
74 }
75
76 System.out.println("Winner of the League is " + currBestTeam
77
78 }
```

- b. Run the application by clicking the Debug button.



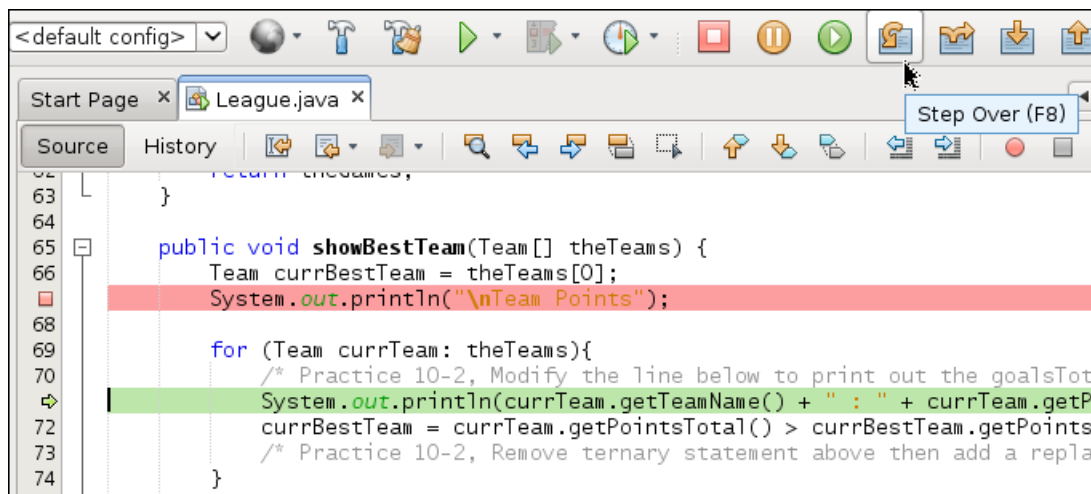
- c. The code will run until it hits the breakpoint and stop there. Look at the bottom panel. This shows the relevant variables; `this` (the `League` object), `theTeams` (the `Team` array passed into the method), and `currBestTeam` (a local variable in the method).

| Variables x Breakpoints Output |        |                |
|--------------------------------|--------|----------------|
| Name                           | Type   | Value          |
| <Enter new watch>              |        |                |
| ▶ this                         | League | #253           |
| ▶ theTeams                     | Team[] | #254(length=2) |
| ▶ currBestTeam                 | Team   | #255           |

- d. Try expanding the variables. For example, notice how you can see the attributes of the `Team` object currently referenced by `currBestTeam`.

| Variables x Breakpoints Output |          |                |
|--------------------------------|----------|----------------|
| Name                           | Type     | Value          |
| <Enter new watch>              |          |                |
| ▶ this                         | League   | #253           |
| ▶ theTeams                     | Team[]   | #254(length=2) |
| ▼ currBestTeam                 | Team     | #255           |
| ▶ teamName                     | String   | "The Greens"   |
| ▶ playerArray                  | Player[] | #264(length=3) |
| ▶ pointsTotal                  | int      | 2              |

- e. Click Step Over (F8) twice.

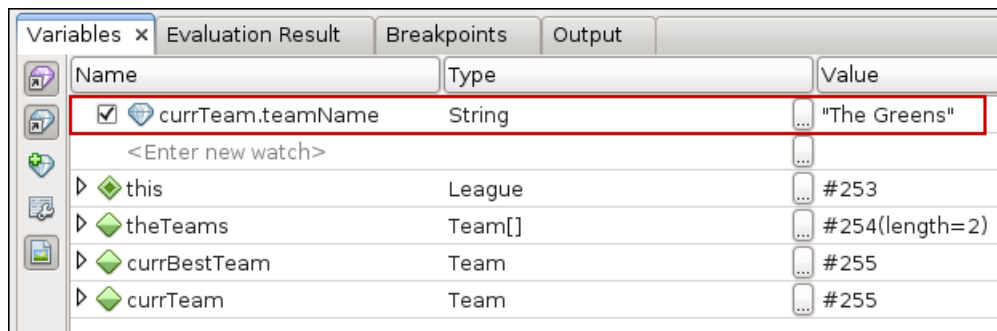
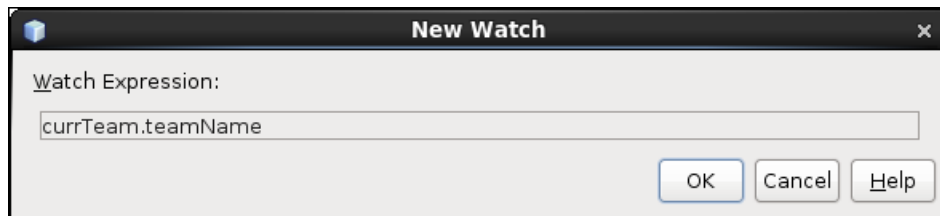
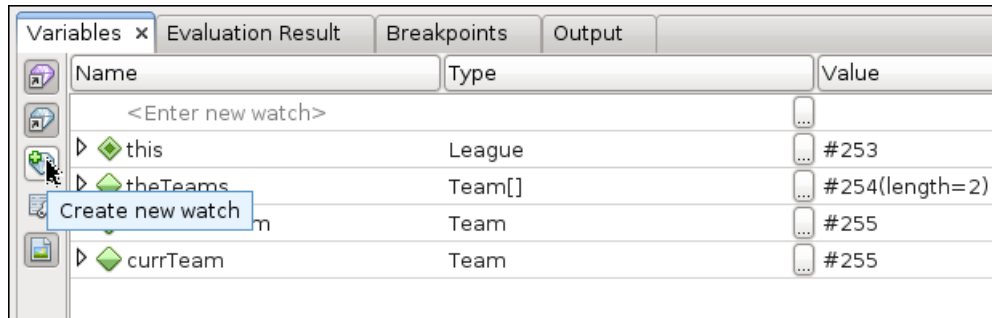


- f. The program is now within the **for** loop and you see a new variable, `currTeam`, in the debugging pane.

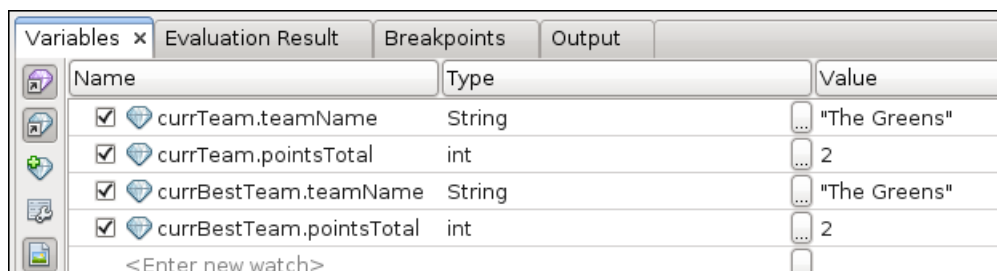
Because the code next does a comparison between the `pointsTotal` attribute of `currTeam` and `currBestTeam`, you might want to look at these values as you step through the code. However, to do that, you would have to expand these variables each

time to see what they contain. Fortunately, NetBeans allows you to specify certain variables to watch, called Watch variables.

3. Set Watch variables for `currTeam.pointsTotal` and `currBestTeam.pointsTotal`.
  - a. Click Create New Watch and, in the dialog box, type in `currTeam.teamName`. Then click OK. Notice that the team name for the Team referred to by `currTeam` is now displayed.



- b. Set up three more Watch Variables for `currTeam.pointsTotal`, `currBestTeam.teamName`, and `currBestTeam.pointsTotal`. (The values for `currTeam.pointsTotal` and `currBestTeam.pointsTotal` may be different on your machine.)



- c. Remember that, on this first iteration, `currTeam` and `currBestTeam` point to the Team at element 0 of the array `theTeams`.

- d. Use F8 (Step Over) to single step through each line in the **for** loop. Keep going until the green line (showing the line being executed) jumps back up to the start of the **for** loop.

Notice that there is now no `currTeam` variable, because it is a local variable that only exists inside the loop.

"currTeam" is not a known variable in the current context..

- e. Stop the run of the debugger by clicking the stop icon.



- f. Start debugging again and check the value of `currBestTeam.pointsTotal`. If the value is 3 or less than 3, continue with (g) below; otherwise, stop and start the debugger until `currBestTeam.pointsTotal` is 3 or less than 3. (You are doing this because you want the value of `currBestTeam.pointsTotal` to be 3 or less than 3 because that means it cannot possibly be the winning team and, therefore, you will see it change as you iterate through the loop.)
- g. Use F8 to iterate through the loop. As you do this, notice how `currBestTeam` changes when it is compared against a `currTeam` with better `pointsTotal`.
- h. Notice that, when it gets to the end of the **for** loop, it will either iterate again or exit. After it exits the loop, you can click Continue (the green arrow) or press F5.



The application will now complete, although it will not automatically go to the Output screen.

- 4. Add code to the `showBestTeam` method to differentiate between teams that have the same number of points. You will make the determination based on the total number of goals scored by the team.

You will add a new field to `Team` to store the goals total, and then modify the code in the `getDescription` method of `Game` so that it will increment each team's total number of goals as it iterates through the `Game` array.

- a. Just as you did for the `pointsTotal` field in `Team`, add a new `goalsTotal` `int` to `Team`, and use NetBeans to create getter and setter methods for it (use the Refactor utility).

```
private int goalsTotal;
```

- b. As you did when creating the `pointsTotal` attribute, add an `incTotalGoals` method.

```
public void incGoalsTotal(int goals){
 this.goalsTotal = this.goalsTotal + goals;
}
```

- c. Incrementing the `goalsTotal` for each goal scored requires a minor change to `getDescription` in the `Game` class. First find the **if/else** clause that increments the

goals in a game (and is used to determine who won the game). The code looks like this:

```
if (currGoal.getTheTeam()==homeTeam) {
 homeTeamGoals++;
}
else {
 awayTeamGoals++;
}
```

- d. Now just add a line to increment the `Team` object with each goal scored by each team. It will now look like this (newly added code bolded).

```
if (currGoal.getTheTeam()==theTeams[0]) {
 homeTeamGoals++;
 theTeams[0].incGoalsTotal(1);
}
else {
 awayTeamGoals++;
 theTeams[1].incGoalsTotal(1);
}
```

5. Add code to the `showBestTeam` method of `League` so that, if two teams have the same number of points, you check to see which has the most total goals.
- a. Now that you need a more complex ***if/else*** if block, rewrite the ternary statement in the `showBestTeam` method in the form of an ***if*** statement. (The code below replaces the single line ternary statement.)

```
if (currTeam.getPointsTotal() >
 currBestTeam.getPointsTotal()) {
 currBestTeam = currTeam;
}
```

- b. After the ***if*** block that you just added, add an ***else if*** to test whether `currTeam.pointsTotal` and `currBestTeam.pointsTotal` are equal. (New code is bolded.)

```
if (currTeam.getPointsTotal() >
 currBestTeam.getPointsTotal()) {
 currBestTeam = currTeam;
}
else if (currTeam.getPointsTotal() ==
 currBestTeam.getPointsTotal())
```

- c. Within the ***else if***, check to see whether `currTeam` scored more total goals than `currBestTeam` and, if they did, make `currBestTeam` now reference `currTeam`. The entire ***else if*** clause now looks like this:

```
else if (currTeam.getPointsTotal() ==
```



```
currBestTeam.getPointsTotal()) {
 if (currTeam.getGoalsTotal() >
 currBestTeam.getGoalsTotal()) {
 currBestTeam = currTeam;
 }
}
```

- d. Modify the `System.out.println` statement that prints the `pointsTotal` for each Team so that it now prints the `goalsTotal` as well. (New code is bolded.)

```
System.out.println(currTeam.getTeamName() + ":" +
 currTeam.getPointsTotal() + ":" + currTeam.getGoalsTotal());
```

- e.
6. Test your new code.
- a. Run the project a number of times so that you can see what happens when two teams have the same number of points. Eventually you should see something like this, where (in this case) the Reds win the league because, although they scored the same number of points, they scored more goals.

```
<-- earlier lines of output omitted -->

Team Points
The Greens:3:6
The Reds:3:7
Winner of the League is The Reds
```

- b. If you would like to try some more debugging, add Watch Variables for `currTeam.goalsTotal` and `currBestTeam.goalsTotal` and single-step through the code so you can see what is happening as the **if** block executes.

Of course there is still a problem if two teams score the same points total and the same goals total. How might you address this? One way would be to add another **else if**, this time to check for goals conceded. Or the code could allow for more than one winner, or perhaps put all team names with the same score into a playoff set of games. But for now, this is all you will do. In practices 13-2 and 13-3, you will create more elegant code to list all the teams in rank order.

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.



**Practices for Lesson 11:  
Working with Arrays, Loops,  
and Dates**

For Instructor Use Only.  
This document should not be distributed.

## Practices for Lesson 11: Overview

---

### Practices Overview

In these practices, you will use an ArrayList to iterate through data.

For Instructor Use Only.  
This document should not be distributed.

## Practice 11-1: Iterating Through Data

---

### Overview

In this practice, you will write code to allow teams of any size to be created from a comma-separated list of names stored in a `String`. You will find a new `PlayerDatabase` class in the `utility` package. At the moment, it contains only a `String` with a comma-separated list of names.

### Tasks

1. Close any open code tabs, and open the **11-ArraysLoopsDates\_Practice1** project.
2. In the `PlayerDatabase` class in the `utility` package, create an `ArrayList` and populate it with the names in the `String` `authorList`.

- a. Open the `PlayerDatabase` class and declare an `ArrayList` of type `Player`. Name it `players`.

```
private ArrayList <Player> players;
```

- b. Click the red error icon in the margin. You will see that you need to import `java.util.ArrayList`. Do this, and then modify the `java.util.ArrayList` import so that it now imports `java.util.*` (all classes in `java.util`). The import statement will now look like this.

```
import java.util.*;
```

- c. Add another import for `soccer.Player`.

```
import soccer.Player;
```

- d. You need to find some way to iterate through the names and add each to the `ArrayList`. Look up `StringTokenizer` in the Javadocs. Notice that it is also in the `java.util` package.

- e. Create a no-argument constructor for the `PlayerDatabase` class.

```
public PlayerDatabase() {
}
```

- f. Within the constructor, create a `StringTokenizer` `authorTokens` that is built on the `authorList` `String`.

```
StringTokenizer authorTokens =
 new StringTokenizer(authorList, ",");
```

- g. Instantiate the `ArrayList` `players`.

```
players = new ArrayList();
```

- h. Create a **while** loop to iterate through the `StringTokenizer`. On each iteration, add a new `Player` to the `ArrayList`. Notice how easy it is to do this. With an array, you would have to find out the number of players in `authorList` and then add each player to consecutive elements of the array. Using an array is possible, but not as easy as using an `ArrayList`.

```
while (authorTokens.hasMoreTokens()) {
 players.add(new Player(authorTokens.nextToken()));
}
```

Now you have an `ArrayList` of eligible players that you can use to populate teams.

3. Create a method to return an arbitrarily sized team.
- a. Create a method, `getTeam`, that takes an `int` (`numberOfPlayers`) and returns an array of `Players`.

```
public Player[] getTeam(int numberOfPlayers) {
}
```

- b. Within the `getTeam` method, create a `Player` array named `teamPlayers`.  
`Player[] teamPlayers = new Player[numberOfPlayers];`
- c. Now create a **for** loop to iterate through this array.

```
for (int i = 0; i < numberOfPlayers; i++) {
}
```

- d. On each iteration of the loop, randomly select a `Player` from the `players` `ArrayList` and add that player to the `teamPlayers` array.

```
int playerIndex = (int) (Math.random() * players.size());
teamPlayers[i] = players.get(playerIndex);
```

- e. Remove the player just selected from the `players` `ArrayList` (this is to ensure that the same player cannot play for more than one team). Notice how this is easy to do with an `ArrayList`; it would be much more difficult if using an array.

```
players.remove(playerIndex);
```

- f. Just after the **for** loop, return the `teamPlayers` array.

```
return teamPlayers;
```

4. Modify the `createTeams` method to use the `PlayerDatabase` class.

- a. Go to the `createTeams` method of `League` and remove all code that creates a `Player` object or a `Player` array. The following code should remain (do not worry that it has errors).

```
Team team1 = new Team("The Greens", thePlayers1);
Team team2 = new Team("The Reds", thePlayers2);
Team[] theTeams = {team1, team2};
return theTeams;
```

- b. Instantiate a new `PlayerDatabase` object at the start of the `createTeams` method. You will need to import it also.

```
PlayerDatabase playerDB = new PlayerDatabase();
```

- c. Modify the lines that instantiate `team1` and `team2` so that they now use `playerDB` for the players. The lines will now look like this.

```
Team team1 = new Team("The Greens", playerDB.getTeam(3));
Team team2 = new Team("The Reds", playerDB.getTeam(3));
```

- d. Run the application a few times to test it. It should work as before, except now the players will be randomly assigned to each team.
5. Make the `createTeams` method more general-purpose by passing in team names and team sizes.

- a. Change the `createTeams` method signature to receive a `String` with the team names and an `int` for the number of players in each team.

```
public Team[] createTeams(String teamNames, int teamSize) {
```

- b. Because the team names will be passed in as a comma-separated list, you must (as before) use a `StringTokenizer` to set up a **for** loop to iterate through however many teams have been specified. Create the `StringTokenizer` now (you may need to click the red dot to import it). Put this line just below the line that instantiates the `PlayerDatabase` object.

```
StringTokenizer teamNameTokens = new
 StringTokenizer(teamNames, ",");
```

- c. Create a `Team` array called `theTeams`. It will have one element for each team name passed in.

```
Team[] theTeams = new Team[teamNameTokens.countTokens()];
```

- d. Write a **for** loop that iterates through the array and creates a new `Team` for each element. You can use the `StringTokenizer` method `nextToken` to get the team name, and the `PlayerDatabase` method `getTeam` to get the array of type `Player`.

```
for (int i = 0; i < theTeams.length; i++){
 theTeams[i] = new Team(teamNameTokens.nextToken(),
 playerDB.getTeam(teamSize));
}
```

- e. Remove the remainder of the method except for the return statement.

6. Modify the call to `getTeams` in the `main` method of `League` to pass in team names and team size.

- a. Replace the current call to the `createTeams` method with the following:

```
Team[] theTeams = theLeague.createTeams("The Robins,The
Crows,The Swallows", 3);
```

- b. Run the application a few times. It should work as before.
- c. The method `createGames` is currently hard-coded; otherwise you could change the number of teams by changing the call to `createTeams`. However, you can change team size, so try making your league 5 per side.

## Rewrite `createGames` to Generate All-Play-All Set of Games

7. Create a nested loop in `createGames` to return an array of Games that ensures that all teams play each of their competitors.

- Delete everything in the `createGames` method except the return statement.
- Instantiate an `ArrayList` to hold the games that you will create. (You may need to import `ArrayList`.)

```
ArrayList<Game> theGames = new ArrayList();
```

- Create a **for** loop to iterate through all the teams in the `Team` array.

```
for (Team homeTeam: theTeams) {
}
```

- For each `Team` you need to create a `Game` matching that `Team` against one of their competitors. Therefore, create another **for** loop within the one you just created. Use `awayTeam` as the local variable name this time.

```
for (Team awayTeam: theTeams) {
}
```

- All you need to do now is to create a `Game` for each iteration of the inner loop. However, that means that “The Crows” could end up playing “The Crows.” Therefore, write an **if** statement to exclude this possibility. The entire nested loop will look like this.

```
for (Team homeTeam: theTeams) {
 for (Team awayTeam: theTeams) {
 if (homeTeam!=awayTeam) {
 theGames.add(new Game(homeTeam, awayTeam));
 }
 }
}
```

- Finally, you need to return an array, not an `ArrayList`. Therefore, you must use the `toArray` method of `ArrayList`. Just use the following code; it will be explained later:  

```
return (Game[]) theGames.toArray(new Game[1]);
```
- Test the application. It should work as before except that there are now more games than before (the Swallows get a chance!). The way it is set up now, teams play each other twice, once at home and once away.

This is the end of this practice. Shut down any NetBeans tabs that contain Java code.



## Practice 11-2: Working with LocalDateTime

---

### Overview

In this practice, you work with the `LocalDateTime` object so that games have a `LocalDateTime` attribute.

### Tasks

1. Open the **11-ArraysLoopsDates\_Practice2** project in NetBeans.
2. Add a new attribute to the `Game` object.
  - a. Add a `LocalDateTime` attribute, `theDateTime`, just below the `goals` attribute.  

```
private LocalDateTime theDateTime;
```
  - b. You will see an error because this class is not in `java.lang`. Add the `java.time.*` package as an import by clicking the red dot and selecting the first option Add import...
  - c. You will see that the import is only for `LocalDateTime`. Therefore, replace `LocalDateTime` in the import statement with a `*` (now all classes in this package will be available to you). The import statement will now look like this:  

```
import java.time.*;
```
  - d. Use the NetBeans refactor feature to create getter and setter methods for this attribute.
  - e. Modify the constructor of the `Game` class to set this `LocalDateTime` attribute. The constructor will now look like this (new code is bolded):

```
public Game(Team homeTeam, Team awayTeam,
 LocalDateTime theDateTime) {
 this.homeTeam = homeTeam;
 this.awayTeam = awayTeam;
 this.theDateTime = theDateTime;
}
```

3. Modify the `getDescription` method of the `Game` class to work with this new attribute.
  - a. Modify the `getDescription` method of `Game` so that it now returns the date and time of the game. The line that you need to modify currently is:

```
returnString.append(this.getHomeTeam().getTeamName() + " vs. " +
 this.getAwayTeam().getTeamName() + "\n");
```

After you modify it, it will be (with new code in bold):

```
returnString.append(this.getHomeTeam().getTeamName() + " vs. " +
 this.getAwayTeam().getTeamName() + "\n" +
 "Date " +
 this.theDateTime.format
 (DateTimeFormatter.ISO_LOCAL_DATE) + "\n");
```

You may have to import `DateTimeFormatter`.

- b. In the `createGames` method of `League`, modify `theGames.add` method inside the `if` block to pass a new `LocalDateTime` object to the constructor of `Game`. Use `LocalDateTime.now()` to instantiate the `LocalDateTime` object (new code is bolded). You will have to import `LocalDateTime`.
- ```
theGames.add(new Game(homeTeam, awayTeam, LocalDateTime.now()));
```
- c. Run the application. You should see the time for the game before the description of the play. (At the moment this is a little strange because the `LocalDateTime` value for all the games is now!)
4. Modify the `createGames` method to increment the date that each game is scheduled to be played.
- a. In the `createGames` method of `League`, add a line at the start of the method to declare an `int` variable, `daysBetweenGames`, and initialize it to 0.
- ```
int daysBetweenGames = 0;
```
- b. At the start of the `if` block (inside the inner loop) add a line to increment the `daysBetweenGames` variable by 7.
- ```
daysBetweenGames += 7;
```
- c. Modify the call to the `Game` constructor so that each `Game` is now scheduled seven days later than the previous one. Look in the Javadocs for `LocalDateTime` to see what method to use (new code is bolded below).
- ```
theGames.add(new Game(homeTeam, awayTeam,
LocalDateTime.now().plusDays(daysBetweenGames)));
```
- d. Run the application again. You should now see that each game is now set for seven days later than the previous game. Of course, now it is a little strange because you also see the game result even though that is in the future! (But remember the random game generator is principally for testing and would not be used in the real world operation of the application).
5. Write a `getLeagueAnnouncement` method that calculates how long the League lasts.
- a. At the bottom of the `League` class, add a new method, `getLeagueAnnouncement`.
- ```
public String getLeagueAnnouncement(Game[] theGames){  
  
}
```
- b. Because you will need the `Period` class, look it up in the Javadocs now. Can you see which method you will need to create a `Period` object? You will need the static method `between` that takes two `LocalDate` parameters and returns a `Period` object.
- c. In addition, you need to deal with the fact that the attribute you used on `Game` is `LocalDateTime` not `LocalDate`. How can you convert `LocalDateTime` to `LocalDate`? Look in the Javadocs for `LocalDateTime`. Examine the method `toLocalDate`.

- d. Add a line that creates a `Period` object based on the dates of the first and last games. You will need to use NetBeans to import the `java.time` package for `Period`.

```
Period thePeriod =  
    Period.between(theGames[0].getTheDateTime().toLocalDate(),  
        theGames[theGames.length - 1].getTheDateTime().toLocalDate());
```

- e. Use String concatenation (or a `StringBuilder` and the `append` method) to return a String that describes the length of the League tournament. For example:

```
return "The League is scheduled to run for " +  
    thePeriod.getMonths() + " month(s), and " +  
    thePeriod.getDays() + " day(s)\n";
```

6. Test the application.

- a. Just before the **for** loop in the `main` method of `League`, add a `System.out.println` to print the description of the league based on the `getGamesAnnouncement` method.
- ```
System.out.println(theLeague.getLeagueAnnouncement(theGames));
```
- b. Run the application. Scroll up to the top of the output. You will see something like the following.

```
The League is scheduled to run for 1 month(s), and 4 day(s)

The Greens vs. The Reds
Goal scored after 17.0 mins by Rafael Sabatini of The Reds
Goal scored after 32.0 mins by Robert Service of The Reds
Goal scored after 35.0 mins by Geoffrey Chaucer of The Greens
The Reds win (1 - 2)

The Reds vs. The Greens
Goal scored after 21.0 mins by Rafael Sabatini of The Reds
Goal scored after 24.0 mins by George Eliot of The Greens

<-- Further output omitted -->
```

- c. Try changing the value of the `daysBetweenGames` variable to see whether the length of time required to run the league changes.

This is the end of this practice. Shut down any tabs containing Java code.



## **Practices for Lesson 12: Using Inheritance**

## Practices for Lesson 12: Overview

---

### Practices Overview

In these practices, you will work with class hierarchies. Remember that class hierarchies give you a way of not only sharing like behaviors in different classes, but they also allow you to use a common reference type for a set of objects of related but different types. In the next lesson, you will see how powerful this can be.

For Instructor Use Only.  
This document should not be distributed.

## Practice 12-1: Creating a Class Hierarchy

---

### Overview

In this practice, you will rewrite the `playGame` method. Now that you have learned how to use the **for** loop with an increment, you can iterate through a game from the beginning until the end 90 minutes later. This way, you can add various events at the point in the game when they take place. The code you write here will replace `utility.GameUtils`.

1. Open the **12-UsingInheritance\_Practice1** project.
2. Modify the call to the current `playGame` method.
  - a. Open the `League` class and, within the **for** loop in the `main` method, comment out the `System.out.println` statement that calls the `getDescription` method.
  - b. In the same method, add a `break` command after the call to the `playGame` method. This will ensure that only one game is played and that no description is generated.
  - c. Comment out `theLeague.showBestTeam` just below the **for** loop.
  - d. Check that the loop and line just below looks like this (your changes—the `/`'s and the `break` statement—are shown bolded).

```
for (Game currGame: theGames) {
 currGame.playGame();
 break;
 //System.out.println(currGame.getDescription
}
//theLeague.showBestTeam(theTeams);
```

- e. Open the `Game` class and remove the line in the no parameter `playGame` method. The `playGame` method that takes an `int` (`maxGoals`) can remain for now, because you will be able to copy some code from it. The no parameter `playGame` method should now look like this:

```
public void playGame() {
}
```

3. Rewrite the `playGame` method to add goals at various (randomly generated) intervals in a game.
  - a. Within the (now empty) `playGame` method, add a **for** loop that iterates through the game where `i` (the index) represents the minutes passed in the game. Put a `System.out.println` inside the loop to tell you the minutes passed.

```
for (int i = 1; i <= 90; i++) {
 System.out.println(i);
}
```

- b. Run the application. You should see a printout of the index from 1 to 90 in the console (plus the league announcement and results).

```
1
2
...< lines omitted>...
89
90
```

- c. Because you will be adding `Goal` objects randomly as you iterate through the loop, an `ArrayList` will be very useful. Create an `ArrayList` inside the `playGame` method but before the `for` loop. Use the `<Goal>` notation to ensure that its type is `Goal`. (You will also need an import for `ArrayList`.)

```
ArrayList <Goal> eventList = new ArrayList();
```

Note the use of `eventList` as the name for the `ArrayList` of type `Goal`. This is because later you will modify your code to work with different events and not just goals.

- d. Just below this, declare a `Goal` named `currEvent` (again, `currEvent`, not `currGoal`).

```
Goal currEvent;
```

- e. Within the **`for`** loop, add an **`if`** block to (randomly) add a `Goal`. You should set the randomness so that you do not have an unreasonable number of goals (try testing for the double returned by `Math.random()` being greater than 0.95).

```
if (Math.random() > 0.95) {
}
```

- f. Move the `System.out.println` statement so that it is within the `if` block as shown below:

```
if (Math.random() > 0.95) {
 System.out.println(i);
}
```

- g. Now run the application a few times so you can get a feel for the likely number of goals per game. You should see that the number of goals is rarely more than 8 (probably realistic) but feel free to change the probability if you would like. (Note that the numbers will represent the minute in which the goal is scored.)

```
17
29
55
71
```

- h. Comment out the `System.out.println` statement.
- i. Immediately below the `System.out.println` statement, add code to create a new `Goal`.

```
currEvent = new Goal();
```



- j. Set the `Team` for that `Goal`. Notice how convenient the ternary operator is for this purpose.

```
currEvent.setTheTeam(Math.random() > 0.5?homeTeam:awayTeam);
```

- k. Set the `Player` for that `Goal`. Notice how `(int) Math.random()` is used to select which player on that team is chosen.

```
currEvent.setThePlayer(currEvent.getTheTeam().
 getPlayerArray()[(int) (Math.random() *
 currEvent.getTheTeam().getPlayerArray().length)]);
```

- l. Set the time attribute for `currEvent`. This will be the index, `i`.

```
currEvent.setTime(i);
```

- m. Add the `Goal` to the `eventList` `ArrayList`.

```
eventList.add(currEvent);
```

- n. To complete the method, you must copy the `Goal` references in the `ArrayList` to an array. Therefore, you need to create the array (and now you know the size based on the size of the `ArrayList`), and then use the `toArray` method of `ArrayList` to copy over the elements from the `ArrayList` to the array. Here are the two lines—they need to go at the end of the method, after the end of the **for** loop.

```
this.goals = new Goal[eventList.size()];
eventList.toArray(goals);
```

4. Test your new `playGame` method.

- In the `main` method of the `League` class, uncomment the lines that you had previously commented out (`System.out.println` and the call to the `showBestTeam` method), and remove the `break` statement.
- Run the application. It should work exactly as it did previously.
- Remove the `playGame` method that takes the parameter `(int maxGoals)`. Be careful not to remove the one you just created!
- Delete the `GameUtils` class in the utility package. You will not need it any more.

This is the end of this practice. Close any open Java code tabs.

## Practice 12-2: Add a GameEvent Hierarchy

---

### Overview

In this practice, you will make the application more flexible by adding other event types (currently there is only the `Goal` class). You already have `Goal` as one type of event, and you will create `Possession` as another type of event. `Possession` will be used to represent the time that one or other of the teams has possession of the ball and as such would need to store `Team` and `Player` just like `Goal` does. Both `Goal` and `Possession` will extend a new abstract `GameEvent` class that you will also create.

### Tasks

1. Open the **12-UsingInheritance\_Practice2** project.
2. It is time to create an event type hierarchy. In this new hierarchy, `Goal` will be one of many game events that are recorded.
  - a. Create a `GameEvent` class in the `soccer` package. Make it abstract, because a `GameEvent` object will not be instantiated—only `GameEvent` subclasses like `Goal`, `Kickoff`, `Pass`, `Dribble`, `Tackle`, `Foul`, and similar objects will be instantiated. You will need to add the `abstract` modifier to the class added by NetBeans. The class declaration will look like the following:

```
package soccer;

public abstract class GameEvent {

}
```

- b. Modify the `Goal` class so that it extends `GameEvent`. The class declaration in `Goal` will now be:

```
public class Goal extends GameEvent {
...<lines omitted>...
}
```

- c. Look at the code in `Goal`. The attributes of `Goal` (and their getters and setters) will be required for all `GameEvent` types, so all of the code currently in `Goal` can be moved up to the superclass, `GameEvent`. Press **Ctrl + X** to cut all the code in `Goal` (not including the class declaration), and press **Ctrl + V** to paste it into the new `GameEvent` class.
- d. Run the application to make sure that it still works exactly as before.

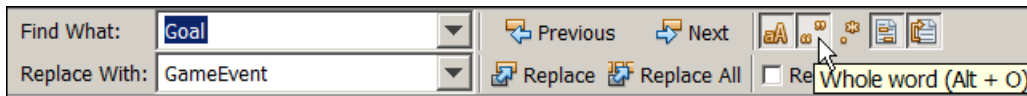
3. You need to modify all the code in your application so that instead of working with `Goal` references, it works with `GameEvent` references. The only code that will still use `Goal` will be the creation of the `Goal` object.

**Note:** Follow the instructions very carefully in this step.

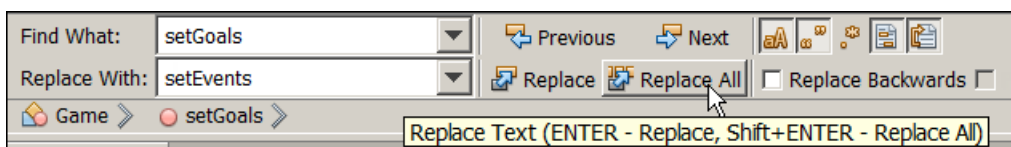
- a. Go to the `Game` class and find the following line:

```
private Goal[] goals;
```

- b. You need to replace all occurrences of `Goal` with `GameEvent`. Click `Edit > Replace`. In the Find What field, enter `Goal`, and in the Replace With field, enter `GameEvent`. Make sure to check `Whole Word`.



- c. Click `Next`, and then click `Replace` for each occurrence of `Goal` **EXCEPT** for the occurrence immediately after **new**, where a `Goal` object is instantiated. On this occasion, click `Next`, then continue to replace all subsequent occurrences of `Goal` with `GameEvent`. In particular, make sure that the instantiation of a `Goal` array does change to the instantiation of a `GameEvent` array.
- d. Double-check to ensure that there are no errors. For example, if you accidentally replaced `new Goal()` with `new GameEvent()` this will cause an error because `GameEvent` is abstract and cannot be instantiated. Note that there should be only one occurrence of `Goal` in the code.
- e. Run the project. It should run fine but the text will report goals as “GameEvent.”. You will improve the output later.
- f. As you did replacing `Goal` with `GameEvent`, use `Find/Replace` to find and replace the following text in the `Game` class. You can use **Replace All** for these, but make sure that `Whole Word` is still checked. Note that these changes are only for readability and do not affect the functionality of the code.



| Find     | Replace    |
|----------|------------|
| goals    | gameEvents |
| currGoal | currEvent  |
| getGoals | getEvents  |
| setGoals | setEvents  |

4. Now you can add a new type of event. You will add a `Possession` class that will represent which team (and player) has possession of the ball at various times in the game.
- a. Create a new class called `Possession`. Modify its signature so that it extends `GameEvent`.

```
public class Possession extends GameEvent {
}
```

**Technical discussion:**

At this point, you may note that both `Possession` and `Goal` have no code in them and are essentially the same, other than that they represent different events in a soccer game. However, many enhancements could be made to either one that could make them very different. In the next practice, you will add a method to each subclass that has a different implementation.

- b. Go to the `playGame` method of `Game`. Replace the current line that instantiates a `Goal`. Use `Math.random()` with a ternary operator to determine whether the event should be `Goal` or `Possession`.
- ```
currEvent = Math.random() > 0.6 ? new Goal() : new Possession();
```
- c. Modify the `getDescription` method of `Game` so that, instead of printing "GameEvent scored after" for each event, it prints the `currEvent` object. You can do this by modifying the text "GameEvent scored after" that is appended to `returnString` to the bolded text below (the entire `append` method call is shown).

```
returnString.append (currEvent + "after " +  
    currEvent.getTime() + " mins by " +  
    currEvent.getPlayer().getPlayerName() +  
    " of " + currEvent.getTeam().getTeamName() +  
    "\n");
```

- d. Run the application. What do you see? `System.out.println` is calling the `toString` method on `currEvent` and, because `currEvent` does not have a `toString` method, it is being called on `Object` (the parent of all objects). The `toString` method on `Object` tells us the name of the class (fully qualified plus a hex value). This is not very useful to a user; therefore, in the next practice you will see how to override the `toString` method on `Object` in each of the `GameEvent` subclasses. You may also notice that `Possession` `GameEvents` are being counted as goals. You will deal with both of these issues in the next practice.

This is the end of this practice. Close any open Java code tabs.

Practices for Lesson 13: Using Interfaces

Practices for Lesson 13: Overview

Practices Overview

In these practices, you will work with class hierarchies and interfaces. Interfaces do not allow code reuse of methods as do superclasses, but they do allow you to use a common reference for a group of related classes, thus allowing polymorphic code. Interfaces also are not part of the class hierarchy; therefore, classes in different hierarchies can implement the same interface.

For Instructor Use Only.
This document should not be distributed.

Practice 13-1: Overriding the `toString` Method

Overview

In the practice for the previous lesson, you created a new class hierarchy with a `GameEvent` superclass and two subclasses `Possession` and `Goal`. One of the advantages of this approach is that common code can be put in the superclass and code specific to either `Possession` or `Goal` can be put in the class itself. Then, by using `GameEvent` as the reference to objects of either subclass, polymorphism ensures that the method will be called on the actual object.

Tasks

1. Open the **13-UsingInterfaces_Practice1** project.
2. Implement the `toString` method on `Possession` and `Goal`.
 - a. Run the application. What do you see? `System.out.println` is calling the `toString` method on `currEvent` and, because `currEvent` does not have a `toString` method, it is being called on `Object` (the parent of all objects). The `toString` on `Object` tells us the name of the class (fully qualified plus a hex value). Because this is not very useful to a user, you will override the `Object`'s `toString` method on `Possession` and `Goal`.
 - b. Add the following method to the `Possession` class:

```
public String toString() {  
    return "Possession";  
}
```

- c. Add the following method to the `Goal` class:

```
public String toString() {  
    return "Goal scored";  
}
```

- d. Run the application again. This time the output should be more readable—something like what is shown below:

```
<-- earlier output omitted -->  
  
The Swallow vs. The Crows  
Date: 2014-4-30  
Possession after 5.0 mins by Oscar Wilde of The Greys  
Goal scored after 61.0 mins by William Makepeace Thackeray of  
The Greys  
Goal scored after 84.0 mins by William Makepeace Thackeray of  
The Greys  
The Greys win! (3 - 0)  
  
<-- later output omitted -->
```

You have now written two simple methods that are called polymorphically!

3. Modify the `getDescription` method in the `Game` class so that it correctly totals only `Goal` objects as goals.
 - a. Look at the output again. Is there anything wrong with it? Yes, a `Possession` event is being counted as a goal.
 - b. Find the **`for`** loop in the `getDescription` method. Within this loop, find the **`if/else`** block that checks which `Team` is responsible for a `GameEvent`. Wrap this **`if/else`** block in a new **`if`** statement that checks the `GameEvent` type (shown bolded below).

```
if (currEvent instanceof Goal) {
    if (currEvent.getTheTeam()==homeTeam) {
        homeTeamGoals++;
        homeTeam.incGoalsTotal(1);
    }
    else {
        awayTeamGoals++;
        awayTeam.incGoalsTotal(1);
    }
}
```

- c. Retest the application. Goals should now be identified correctly. However, because `Goal` or `Possession` are only being chosen 95% of the time, games are rather low scoring.
- d. Go to the `playGame` method. Modify the **`if`** clause within the **`for`** loop so that the **`if`** expression is true when a number greater than 0.8 is randomly chosen, and modify the ternary operator to make goals a little less likely, say also 0.8.
- e. Retest the application. It should now show more realistic game scores.
- f. Have a look at the output. You should be able to see the “story” of the game. The `Possession` object and its attributes show where a player passes the ball to another player or where the ball is lost to the opposing side. Now you can see how it would be relatively easy to extend the application to include other events (for example, `Penalty`, `Throw-in`, `FreeKick`, `Corner`, and any other events relevant to a game of soccer).

This is the end of this practice. Close any open Java code tabs.

Practice 13-2: Implementing an Interface

Overview

In this practice, you implement the `Comparable` interface so that you can order the elements in an array.

Tasks

1. Open the **13-UsingInterfaces_Practice2** project.
2. Implement the `Comparable` interface for the `Team` class.
 - a. In Javadocs, look up the `sort` method of the `Arrays` class. Notice that it is an overloaded method. Because the arrays that you have worked with have various kinds of objects in them (`Player`, `Team`, and `Goal`), look at the `sort` method that takes an `Object` array (`sort(Object[] a)`).

Click the link to the `Comparable` interface. In order for the `sort` method to be able to determine the correct order for, say, the `Team` array, you must implement the `Comparable` interface in the `Team` class.

- b. Add `implements Comparable` to the class declaration for `Team`. It will now look like the following (new text bolded). Notice that NetBeans will show an error because you have not yet implemented the `compareTo` method.

```
public class Team implements Comparable {
```

- c. Create the `compareTo()` method of the comparable interface. Notice that it takes one parameter—the object being compared to the `this` object.

```
public int compareTo(Object theTeam) {  
}
```

- d. Declare an `int` that will hold the return value and initialize it to `-1`. You will see an error as you do not have a return statement yet.

```
int returnValue = -1;
```

- e. Add an ***if*** statement block that does the comparison between the "this" `Team`, and the one passed into the method. If the `Team` represented by the "this" reference has fewer points than the `Team` passed in, you should set `returnValue` to `1`. Otherwise the `returnValue` will be equal to `-1` (its initialized value). (The code below will show an error, but do not worry; you will deal with this later.)

```
if (this.getPointsTotal() < theTeam.getPointsTotal()) {  
    returnValue = 1;  
}
```

For Instructor Use Only.
This document should not be distributed.

- f. Add a `return` statement at the bottom of the method.

```
return returnValue;
```

Technical discussion:

Notice that there is still an error in the method. Why do you think the `getPointsTotal` method can be found on the `"this"` object, but not on `theTeam` object? The answer is that `theTeam` is not a `Team` reference, it is an `Object` reference. And even though it is referencing a `Team` object, it cannot access the `getPointsTotal` method because there is no such method on the reference (of `Object` type).

- g. You need to cast the `Object` reference `theTeam` to a `Team` reference. You do this by putting `(Team)` in front of the `theTeam` reference. However, you need to ensure that the casting command and `theTeam` are also parenthesized, so that the casting is on the `theTeam` reference, and not on whatever is returned from the `getPointsTotal` method.

```
if (this.getPointsTotal() < ((Team) theTeam).getPointsTotal()) {  
    returnValue = 1;  
}
```

- h. Go to the `showBestTeam` method of `League`. If you now insert an `Arrays.sort()` command at the beginning of the method, the teams will be listed in rank order. (You will need to import `java.util.Arrays`.)

```
Arrays.sort(theTeams);
```

- i. Run the application. You should see the teams listed in rank order by the number of points. However, if you run the application a few times, you will notice that sometimes when two teams scored the same number of points they are not then ordered by goals scored. For example, given the scores below, the Crows should be listed first.

```
Team Points  
The Swallows:4:3  
The Crows:4:6  
The Robins:4:5  
Winner of the League is The Crows
```

3. Enhance the `compareTo` method in `Team` so that it compares goals scored in the situation when points scored are equal.

- a. In the `compareTo` method in `Team`, add an ***else if*** that deals with the situation when both teams have the same number of points. Inside this ***else if***, add another ***if*** that compares the number of goals scored by each team. The ***if*** block will now look like this (new code bolded):

```
if (this.getPointsTotal() < ((Team) theTeam).getPointsTotal()) {  
    returnValue = 1;  
} else if (this.getPointsTotal() ==  
    ((Team) theTeam).getPointsTotal()) {  
    if (this.getGoalsTotal() < ((Team) theTeam).getGoalsTotal()) {  
        returnValue = 1;  
    }  
}
```

- b. Run the code a few times so that you can see that the ordering is correct.
4. Now most of the code in the `showBestTeam` method is unnecessary and can be removed. In general, you do not want different blocks of code that do the same or similar things.
- a. Remove all the conditional code in the `showBestTeam` method. This includes the ternary statement and the entire ***if*** block.
- b. Run the project a few times to ensure that it is working correctly. Notice that the winner is still printed correctly because after sorting, the `Team` at index zero of the array is the winner.

This is the end of this practice. Close all open Java code tabs.

Practice 13-3: Using a Lambda Expression for Sorting (Optional Practice)

Overview

In this practice, you order the players based on their goal scoring. However, instead of having `Player` implement `compareTo`, you will use a lambda expression.

Tasks

1. Open the **13-UsingInterfaces_Practice3** project.
2. Modify the `Player` class to support ordering players by goals scored.
 - a. Add an `int goalsScored` to `Player`.

```
private int goalsScored;
```
 - b. Use NetBeans to add getter and setter methods for `goalsScored`.
 - c. Add a method `incGoalsScored` that increments the goals scored by a player.

```
public void incGoalsScored() {  
    this.goalsScored++;  
}
```
3. Modify the `getDescription` method in `Game` to increment `goalsScored` for a player when they score. At the end of the `if` block that tests whether `currEvent` is an instance of a `Goal`, add:

```
currEvent.getThePlayer().incGoalsScored();
```

4. Create a new method, `showBestPlayers`, in the `League` class.
 - a. Add a method, `showBestPlayers`, that displays players in order by how many goals they scored.

```
public void showBestPlayers(Team[] theTeams) {  
  
}
```

- b. Write code to create a single `ArrayList` containing all the players in the various teams. There are a number of ways to do this. Here is some code that uses the `addAll` method of `ArrayList` and the `asList` method of `Arrays`:

```
ArrayList <Player> thePlayers = new ArrayList();  
for (Team currTeam: theTeams){  
    thePlayers.addAll(Arrays.asList(currTeam.getPlayerArray()));  
}
```

- c. Add a `System.out.println` to print a heading.

```
System.out.println("\n\nBest Players");
```

For Instructor Use Only.
This document should not be distributed.

- d. Add a **for** loop that iterates through `thePlayers` `ArrayList` and prints out each player and the goals scored by that player.

```
for (Player currPlayer: thePlayers){
    System.out.println(currPlayer.getPlayerName() + " : " +
        currPlayer.getGoalsScored());
}
```

- e. At the bottom of the main method of `League`, add a call to the `showBestPlayers` method.

```
theLeague.showBestPlayers(theTeams);
```

- f. Run the application. You should see something like this—the players are listed but not yet ranked in order of scoring.

```
<-- output omitted -->

Best Players
Brian Moore : 3
Sean O'Casey : 1
Oscar Wilde : 4
Boris Pasternik : 2
Emile Zola :

<-- output omitted -->
```

5. Write a lambda expression to pass into the `Collections.sort` method.
- You will need the `Collections` class, so first import `java.util.Collections`.
 - Put the following code between the two **for** loops in the `showBestPlayers` method. As mentioned earlier you do not have to implement the `Comparable` interface in the `Team` class. Why? Because with lambda expressions you are passing data to the `sort` method, but you are also passing in functionality.

```
Collections.sort(thePlayers, (p1, p2) ->
    Double.valueOf(p2.getGoalsScored()).compareTo
        (Double.valueOf(p1.getGoalsScored())));
```

- Run the application again. You should now see the players listed correctly based on the number of goals they scored.

This is the end of this practice. Close any open Java code tabs.

Practices for Lesson 14: Handling Exceptions

Practices for Lesson 14: Overview

Practices Overview

In these practices, you will modify your code to catch an exception.

For Instructor Use Only.
This document should not be distributed.

Practice 14-1: Overview – Adding Exception Handling

Overview

In this practice, you will investigate how the code of the soccer application can break under certain circumstances, and modify your code to handle this gracefully.

Tasks

1. Open the **14-Exceptions_Practice1** project.
2. Modify the code in the `League` class so that more teams are requested than can currently be produced from the pool of players.
 - a. Modify the parameters passed to the `createTeams` method in the `main` method of `League` so that four teams are created with eleven players in each.

```
Team[] theTeams =
    theLeague.createTeams
        ("The Robins,The Crows,The Swallows,The Owls", 11);
```

- b. Run the project. What happened? Even though the project built successfully, an **Exception was raised**.

```
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
...at java.util.ArrayList.rangeCheck(ArrayList.java:638)
...at java.util.ArrayList.get(ArrayList.java:414)
...at utility.PlayerDatabase.getTeam(PlayerDatabase.java:36)
...at soccer.League.createTeams(League.java:57)
...at soccer.League.main(League.java:31)
```

This `java.lang.IndexOutOfBoundsException` is a `RuntimeException` and, therefore, unchecked, meaning that you did not have to check for it in your code that called the `getTeam` method of `PlayerDatabase`.

3. Write a **try/catch** block in the `main` method of `League` to catch this Exception.
 - a. After the instantiation of `League`, and before the call to the `createTeams` method, add a `try` statement and an opening left brace.

```
try {
```

- b. Because the entire remainder of the `main` method is dependent on `theTeams` array being populated, place the closing right brace of the `try` block after the last method call in the `main` method.

```
<... code omitted ...>
theLeague.showBestTeam(theTeams);
} // Closing brace for try block here
```

- c. Indent the code within the `try` block to make it more readable.

- d. Add a **catch** block after the closing brace of the **try** block. For now, just catch an `Exception`.

```
catch (Exception e) {  
}
```

- e. Now you will use the `Exception` reference, `e`, to print out a stack trace. Look in Javadocs for the `Exception` method that prints a stack trace—`printStackTrace` looks promising! Notice that it requires a `PrintStream` to be passed to it. Where can you find a `PrintStream`? How about `System.out` or, better yet, `System.err`!
- f. Add a line to print the stack trace of the `Exception` to the error console.

```
e.printStackTrace(System.err);
```

- g. Run the application. Notice that NetBeans prints this to the output window, but in red to indicate that it is using the `PrintStream` in `System.err`.
4. This is all very well, but it is just giving us the default behavior we get if we allow the unchecked `Exception` to pass right up through the call stack. Now we could print out a message telling the user that an `IndexOutOfBoundsException` has been raised and omit the stack trace, but wouldn't it be much better if the user gets a message telling them that there is a specific problem with the `PlayerDatabase` class?
5. Create a custom `Exception` to report the exact problem with `PlayerDatabase`.
- a. Create a class named `PlayerDatabaseException` in the `utility` package that extends `Exception` (and therefore it must be caught).

```
package utility;  
public class PlayerDatabaseException extends Exception {  
}
```

- b. Add a no parameter constructor, and also add a constructor that takes a `String` and calls the constructor on `Exception` (its superclass), passing the `String` message.

```
public PlayerDatabaseException() {}  
public PlayerDatabaseException(String message) {  
    super(message);  
}
```

6. Add code to the `createTeams` method of `PlayerDatabase` to throw the `PlayerDatabaseException` when there is a problem. There are two approaches.
- Try to figure out if the problem will occur. For example, analyze the number of teams requested and the size of each and throw a `PlayerDatabaseException` if there are not enough players available.
 - Check for an `IndexOutOfBoundsException` and, if caught, rethrow as a `PlayerDatabaseException`.

We will use the second approach here.

- a. Look at the `getTeam` method of `PlayerDatabase` of the `utility` package. The problem we witnessed earlier is caused when no players remain in the `ArrayList`.

- b. Add a `try` { immediately before the line that tries to get a `Player` reference from the `ArrayList`, and put the closing brace after the `players.remove()` method call. The try block will now look like this:

```
try {
    teamPlayers[i] = players.get(playerIndex);
    players.remove(playerIndex);
}
```

- c. Add a `catch` block that checks for an `IndexOutOfBoundsException` and, if one is caught, rethrows a new `PlayerDatabaseException` with a suitable message (you will need to import `utility.PlayerDatabaseException`).

```
catch (IndexOutOfBoundsException ie) {
    throw new PlayerDatabaseException("Not enough players in the
    database for the teams requested.");
}
```

- d. Notice that NetBeans shows an error. The `PlayerDatabaseException` is an `Exception` (not a `RuntimeException`) and thus it must be caught. However, because you have already written **try/catch** code in `League`, all you need do here is throw the `PlayerDatabaseException` on up the stack. Modify the `getTeam` method signature as shown below.

```
public Player[] getTeam(int numberOfPlayers) throws
PlayerDatabaseException {
```

- e. Now you will see another error, this time in `League`. You now need to modify the signature of the `createTeams` method in `League` in the same way so that it throws `PlayerDatabaseException` (you will also have to import `PlayerDatabaseException`).
- f. Run the project again. Now the `Exception` will be caught in the `main` method of `League` and will display the following message.

```
utility.PlayerDatabaseException: Not enough players in the
database for the teams requested.
    at utility.PlayerDatabase.getTeam(PlayerDatabase.java:39)
    at soccer.League.createTeams(League.java:57)
    at soccer.League.main(League.java:31)
```

Well done! You have now finished all the practices for this course. The next lesson will discuss some of the approaches used, how they could be improved, and how the application could be extended.

**Practices for Lesson 15:
Deploying and Maintaining
the Soccer Application**

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 15

There are no practices for this lesson.

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 16: Understanding Modules

For Instructor Use Only.
This document should not be distributed.

Practices for Lesson 16: Overview

Overview

In these practices, you will explore how to create Java modular applications by using both the command line and NetBeans.

For Instructor Use Only.
This document should not be distributed.

Practice 16-1: Creating a Modular Application from the Command Line

Overview

In this practice, you create a simple single-class Java application and convert it to a modular application.

Assumptions

You have completed the lecture and reviewed the overview for this practice.

Tasks

1. Login to the Oracle Linux lab environment.
2. Open a terminal in the directory
`/home/oracle/labs/16_ModularSystem/Practices.`
3. Create a project directory, `Prac_16_01_ModularSystem`.

```
mkdir Prac_16_01_ModularSystem
```

4. Change to this directory

```
cd Prac_16_01_ModularSystem
```

5. Your main class will be in the package `com.greetings`. Create a source directory, `src`, and within it the directory structure to match the package name.

```
mkdir -p src/com/greeting
```

6. Within `src/com/greeting` create a new main class, `Main.java`.

- a. Change directories to `src/com/greeting`

```
cd src/com/greeting
```

- b. Create the `Main.java` class

```
touch Main.java
```

- c. Open `Main.java` in `gedit` and add the following code. Remember to save your changes.

```
package com.greeting;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

7. From the `~/labs/16_ModularSystem/Practices/Prac_16_01_ModularSystem` directory, create a `classes` folder to hold the compiled classes. Then compile the `Main.java` class to that folder.

- a. Change directories to the `Prac_16_01_ModularSystem` folder. You can back out of a single directory level entering:

```
cd ..
```

- b. Create the `classes` directory.

```
mkdir classes
```

- c. Compile your code.

```
javac -d classes src/com/greeting/Main.java
```

- d. You can check the file layout by running the `tree` command from the command line. The output should look like this:

```
.
├── classes
│   ├── com
│   │   └── greeting
│   │       └── Main.class
├── src
│   ├── com
│   │   └── greeting
│   │       └── Main.java
```

- e. Test the application from the same location. Java needs to know the location of the class. To do specify this, use the `-cp` (classpath) parameter.

```
java -cp classes com.greeting.Main
```

- f. You should get the following output:

```
Hello World!
```

8. Make this simple application a modular Java application.

- a. Create a module directory in the `src` folder, just above the package folder `com`. Both the module and directory will be named `hello`.

```
cd src
mkdir hello
```

- b. Move `com` to the `hello` directory.

```
mv com hello
```

- c. The file layout should look like this:

```
.
├── classes
│   ├── com
│   │   └── greeting
│   │       └── Main.class
├── src
│   ├── hello
│   │   ├── com
│   │   │   └── greeting
│   │   │       └── Main.java
```

- d. Create a file `module-info.java` and put it in the root directory of the module (parallel with `com`).

```
cd hello
touch module-info.java
```

- e. Open the file in gedit and add the following code. Remember to save your work.

```
module hello{

}
```

Note: This code names the module `hello`. It's also located within the `hello` folder. You'll see in the next practice why this naming consistency is very important.

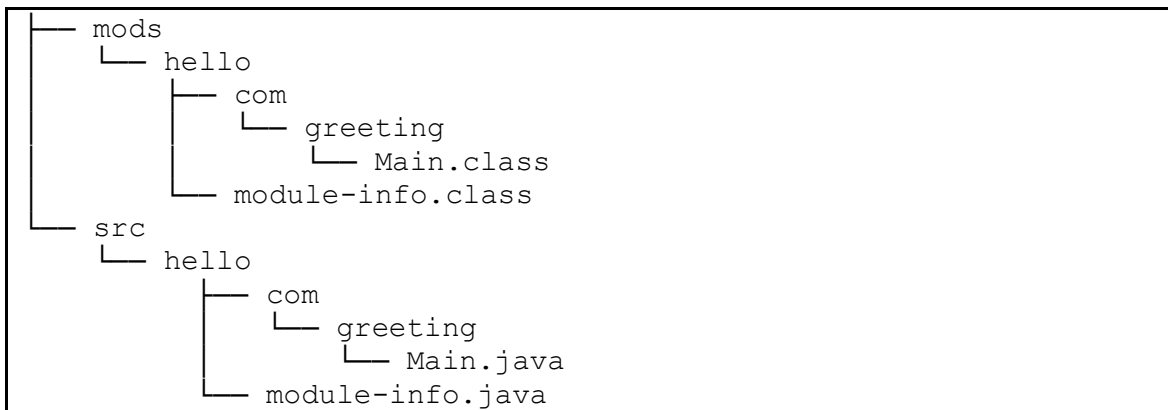
Note: The module is empty. This very simple application uses only one class, `System`, which is included in the `java.base` module. The `java.base` module doesn't need to be explicitly specified as a required module because it is implicitly always present.

- f. Change to the `Prac_16_01_ModularSystem` directory and compile the application.

```
cd ..
cd ..
javac -d mods/hello src/hello/module-info.java
src/hello/com/greeting/Main.java
```

Note: `module-info.java` is specified first so the compiler knows this is modular application and will inform you accordingly if any problems exist with the code. Also note how you must specify the `hello` directory as part of the destination.

- g. Check the directory structure. Your results will look like this:



Note: The `classes` directory is not shown. This directory still exists, but is no longer necessary to run the modular application.

- h. Run your new modular application.

```
java -p mods -m hello/com.greeting.Main
```

- i. You should get the following output:

```
Hello World!
```

Practice 16-2: Compiling Modules from the Command Line

Overview

In this practice, you see a shortcut to compile all modules at once. You don't need to specify each individual module and class for compilation like you saw in the previous practice. For this shortcut to work, it's important to name modules and their directories consistently.

Assumptions

You have completed the lecture, reviewed the overview for this practice, and completed the previous practice.

Tasks

1. Ensure that your terminal is open in the directory
`~/labs/16_ModularSystem/Practices/Prac_16_01_ModularSystem.`

2. Delete your compiled code by removing the `mods` directory.

```
rm -r mods
```

3. Compile all modules at once.

```
javac -d mods --module-source-path src $(find src -name  
"*.java")
```

Note: This automatically creates a directory that is the same as the name given in the `module-info` file.

4. Run the application. It should work.

```
java -p mods -m hello/com.greeting.Main
```

5. In the file explorer, navigate to the `src/hello` directory and open the `module-info` class in `gedit`. Change the name of the class from `hello` to `test`. Remember to save your work.

```
module testjava{  
  
}
```

Note: Now the directory and `module-info` class have different names.

6. Try compiling all modules at again.

```
javac -d mods --module-source-path src $(find src -name  
"*.java")
```

Note: You should get an error message.

```
module test{  
^  
error: cannot access module-info  
    cannot resolve modules  
3 errors
```

7. Close the terminal. The remaining practices are completed in NetBeans.

For Instructor Use Only.
This document should not be distributed.

Practice 16-3: Creating a Modular Application from NetBeans

Overview

In this practice, you create a modular Java application using NetBeans. You'll begin seeing how modules read from each other and how NetBeans allows you to compile many modules at once.

Assumptions

You have completed the lecture, reviewed the overview for this practice, and completed the previous practice.

Tasks

1. Open NetBeans and create a new project for a modular Java application.
 - a. Select **File > New Project**.
 - b. Select Java Modular Project from the list of project types and click Next.
 - c. Name the project `HelloNetBeans`
 - d. Set the project's location to the `Prac_16_03_ModularSystem` folder.
 - e. Ensure that Platform is JDK 11.
 - f. Click Finish.
2. Create the `hello` module within the project.
 - a. Right-click on the project in NetBeans.
 - b. Select **New > Module**.
 - c. Name the module `hello`.
 - d. Click **Finish**.

Note: The `module-info` class is automatically created within the module and displayed for you. Expand the project to see this class in the default package.
3. Create the package `com.greeting` within the `hello` module.
4. Create a new Java Main Class within the `com.greeting` package called `Main`. Add the following code to the class:

```
package com.greeting;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello NetBeans!");
    }
}
```

5. Compile and Run the program. Remember to specify `com.greeting.Main` as the main class. You should get the following output:

```
Hello NetBeans!
```

6. Create a new module called `people`.
7. Create the package `com.name` within the `people` module.
8. Create a new Java Class within the `com.name` package called `Names`. Add the following code to the class. Replace Duke's name with your own:

```
package com.name;

public class Names {
    public static String getName() {
        return "Duke!";
    }
}
```

9. Modify the `module-info` class of the `people` module so that it exports the `com.name` package.

```
module people {
    exports com.name;
}
```

10. Modify the `module-info` class of the `hello` module so that it requires the `people` module.

```
module hello {
    requires people;
}
```

11. Modify the greeting in the `main` method to include the name found in the `Names` class. The print statement will look like this:

```
System.out.println("Hello " + Names.getName());
```

12. You'll notice NetBeans will complain about not finding the `Names` package. In addition to setting up the correct `requires` and `exports` statements, you also need to import the relevant packages into your classes. Add this line of code to the `Main` class.

```
import com.name.Names;
```

13. Build the project.

Note: NetBeans' output window will report that two JARs are created. NetBeans uses an ant script to find and compile all modules associated with the project.

14. Run the project. Your output window will look like this:

```
Hello Duke!
```


Practices for Lesson 17: JShell

Practices for Lesson 17: Overview

Overview

In these practices, you will explore JShell to see how this tool and the REPL process facilitate small code experiments. This includes a look at the NetBeans-integrated version of JShell, which leverages the JShell API.

Don't be afraid to consult the help menu for JShell commands at any time, either to gain insight on how to complete these practices or out of curiosity. JShell as a tool is very good at providing help and feedback. You can access the help menu by typing the `/help` command.

For Instructor Use Only.
This document should not be distributed.

Practice 17-1: Variables in JShell

Overview

In this practice, you will see Read-Evaluate-Print-Loop (REPL) in action by experimenting with variables in JShell. At any time, you can print the full list of all variables you've created by typing `/vars`.

Assumptions

You have completed the lecture and reviewed the overview for this practice.

Tasks

1. Launch JShell.
 - a. Open a terminal or command prompt. On the Oracle University Linux lab machines, this can be done by selecting **Applications > System Tools > Terminal**.
 - b. Enter in the terminal:

`jshell`
2. A few small math experiments can help quickly build your understanding of JShell and REPL process. Start with a simple experiment to step through a single R-E-P-L iteration.
 - a. Type a simple mathematic expression like `2+2`. When you press enter, the expression is **read** into JShell.
 - b. JShell **evaluates** the expression.
 - c. The value of this expression is **printed** to the console. It's also stored as a scratch variable.
 - d. JShell **loops** back to its original state, where you can enter more code snippets.
3. Continue experimenting with mathematical expressions and the different types of variables that can be created in JShell.
 - a. Enter a few numbers. For each number you enter, you'll notice JShell stores the value as a scratch variable.
 - b. Create and name a variable yourself (for example `int x`) and assign it a value.
 - c. Write a math expression that references both a scratch variable and a variable you've named yourself.
 - d. Write a math expression whose answer ends with a decimal. This may include the use of methods from the `Math` class such as `Math.pow` or `Math.sqrt`. JShell evaluates the expression and stores the answer as a variable.
 - e. Create a variable for an object. In this case, a `String`.
 - f. Change the value of a scratch variable.
 - g. Change the value of a variable you've named yourself.
4. Print the list of all variables you've created. JShell reports back the variables' name, value, and type. This is done by typing the `/vars` command.

Practice 17-2: Methods in JShell

Overview

In this practice, you will see how JShell lends itself to writing, testing, and editing methods. At any time, you can get a full list of all the methods you've created by typing the `/methods` command.

Assumptions

You have completed the lecture, reviewed the overview for this practice, and completed the previous practice.

Tasks

1. Continue working in the JShell terminal from the previous practice.
2. Write the method `calcVolume`. This method calculates the volume of a sphere.
 - a. The method is a `void` return type.
 - b. The method accepts a `double` argument `r`. `r` represents the radius of the sphere.
 - c. The volume of a sphere is calculated as $4\pi r^3/3$. The method must print this value.
 - d. This method should require about three lines of code.

Note: Everyone makes mistakes. It's quite easy to forget to add a semicolon or curly brace. Although JShell won't tell you about your mistake until after you think you've finished writing the method, there is a quick way to recall previous lines you've written. Press the up arrow on the keyboard. This will recall the previous line you entered. Continue pressing up to recall even earlier lines.

3. Call the `calcVolume` method to ensure it's working properly. If it's producing the wrong values, you'll need to rewrite the method.
 - a. A radius of 1 should produce a volume of about 4.19.
 - b. A radius of 2 should produce a volume of about 33.51.
4. Write the method `calcArea`. This method calculates the surface area of a sphere.
 - a. The method is a `void` return type.
 - b. The method accepts a `double` argument `r`. `r` represents the radius of the sphere.
 - c. The surface of a sphere is calculated as $4\pi r^2$. The method must print this value.
 - d. This method should require at least three lines of code.
5. Call the `calcArea` method to ensure it's working properly. If it's producing the wrong values, you'll need to rewrite the method.
 - a. A radius of 1 should produce a volume of about 12.57.
 - b. A radius of 2 should produce a volume of about 50.27.

6. The design requirements for your program have suddenly changed! Instead of `calcVolume` and `calcArea` printing their solutions, they must `return` their solution as a `double`. There are two ways to modify an existing method in JShell. We'll examine both.

a. Modify `calcVolume` by rewriting it. This means you'll have to reenter each line of the method, one at a time in JShell.

1) Change the method's return type from `void` to `double`.

2) Change the print statement to a `return` statement.

b. Modify `calcArea` using the JShell edit pad.

1) To open a window for the JShell edit pad, type the following:

```
/edit calcArea
```

2) Change the method's return type from `void` to `double`.

3) Change the print statement to a `return` statement.

4) Press the `Accept` button to save your changes.

5) Press the `Exit` button to exit the edit pad.

c. Call both of these methods in JShell to ensure they are working properly. You'll notice the results are now saved as scratch variables.

Note: You've undoubtedly found the edit pad is a more convenient and precise way to edit methods in JShell. JShell in a terminal is convenient for small tests. But you're starting to see that as testing and editing requirements become more complex, the tools need to become more robust. Imagine how tricky it might be to write an entire class in a JShell terminal without making a single mistake! Thankfully, JShell offers an API that allows its functionality to be incorporated into robust editing tools, like NetBeans.

7. Print the list of all declared methods and their signatures. This is done by entering the `/methods` command.

Practice 17-3: Forward-Referencing

Overview

In this practice, you will see a few more interesting aspects of JShell. This includes forward-referencing, which allows JShell to call methods that haven't yet been written.

You'll write the `earnInterest` method that calculates a balance after interest is earned. This method then calls the `printBalance` method that prints the new balance to the NetBeans output window.

Note: JShell forward-referencing is a little buggy in this version of NetBeans. Please follow this Practice's steps carefully. Alternatively, forward-referencing also works in the JShell standalone.

Assumptions

You have completed the lecture and reviewed the overview for this practice.

Tasks

1. Open a general JShell session in NetBeans.
 - a. Launch NetBeans.
 - b. Select **Tools**.
 - c. Select **Open Java Platform Shell**.
2. Create a `double` variable `interestRate`. Set its value to 0.02 or 2%.

```
[1]-> double interestRate = 0.02;
```

3. Write the `earnInterest` method. This method updates a `balance` argument after interest is earned.
 - a. The method is a `void` return type and accepts an `int` `balance` argument.
 - b. Set the new value of `balance` equal to the old value, plus the accumulated interest rate.

```
[2]-> void earnInterest(int balance) {  
    balance += balance*interestRate;  
    ...  
}
```

- c. The math performed in the previous step should result in a `double`. For this reason, it may be better if the method accepts a `balance` argument as a `double` rather than an `int`. Change the `balance` argument to be a `double` instead of an `int`.

Note: You'll notice it's much easier to make this correction with NetBeans' precision editing than in the JShell standalone. In NetBeans, you simply need to click to position the cursor to the line you want to edit. In the JShell standalone, there's no way to navigate to an earlier part of the snippet after you've pressed enter. You'd either have to write the method over again or open the edit pad once you've finished writing an initial version of the method.

For Instructor Use Only.
This document should not be distributed.

- d. For the final line of this method, pass the updated `balance` to the `printBalance` method. This is considered forward-referencing. You're calling a method that doesn't yet exist.
- e. Press **Enter** following the closing curly brace of the method. You'll notice JShell reports that although the method has been defined, it can't yet be called until the `printBalance` method is also defined.
- f. Your code will look like this:

```
[2]-> void earnInterest(double balance) {  
    balance += balance*interestRate;  
    printBalance(balance);  
}
```

- 4. Write the `printBalance` method. This method outputs a balance to the console.
 - a. The method is a `void` return type and accepts a `double balance` argument.
 - b. On the next line of the method, type `System.` and observe the list of suggestions provided by NetBeans. Another advantage of combining JShell with an IDE like NetBeans is accommodating auto completion. Ensure that this line of code prints the `balance` along with the prefix `"Balance: "`.

Note: Yet another benefit is that NetBeans' JShell retains the NetBeans shortcuts. Simply type **sout + Tab** to create a print statement.

- c. Press **Enter** following the closing curly brace of the method.
- d. Your code will look like this:

```
[3]-> void printBalance(double balance) {  
    System.out.println("Balance: " +balance);  
}
```

- 5. On second thought, it would be better if the output contained a dollar symbol. Add a dollar symbol (\$) to the printout.
 - a. NetBeans' JShell offers a shortcut to retrieve earlier snippets of code. Press the **up-arrow key** and select your snippet for the `printBalance` method.
 - b. Add a dollar symbol to the printout's prefix so that it prints `"Balance: $"`.
 - c. Press **Enter** following the closing curly brace of the method.
 - d. Call the `earnInterest` method. Be sure to provide a double argument.
 - e. Bring up NetBeans' Output window and observe that the balance is printed.
 - f. Call the `earnInterest` method again with a new value of balance and observe the next output.

Note: You can also auto complete like you saw in the JShell standalone. Pressing **up** in NetBeans' JShell shows a list of both commands and lines of code you've recently entered. And if you've already typed a few characters, such as `Str`, pressing **up** reveals a list possible matching data types, methods, and classes.

- 6. Close NetBeans. You've finished the final Practice!

